CS 4310 Operating Systems

Project #2 Simulating Page Replacement Manager and Performance Analysis

Prof. Gilbert Young

Devin Khun

Due Date: December 2, 2025

Important:

- *Please read this document completely before you start coding.*
- *Also, please read the submission instructions (provided at the end of this document) carefully before submitting the project.*

Project #2 Description:

Simulating Page Replacement Manager of the Operating Systems by programming the following three page replacement algorithms that we covered in the class:

(a) First In First Out (FIFO)

(b) Least Recently Used (LRU)

(c) Optimal Algorithm (OPT)

You can use either Java, or C++ for the implementation. The objective of this project is to help student understand how above three page replacement algorithms operates by implementing the algorithms, and conducting a performance analysis of them based on the performance measure of **page faults** for each page replacement algorithm using multiple inputs. Output the details of each algorithm's execution. You need to show what pages are inside the page frames along with the reference string and mark it when a page fault occurred. You can choose your display format, for examples, you can display the results for each reference string in a table format as shown in the class notes. The project will be divided into three phases to help you to accomplish above tasks in in a systematic and scientific fashion: Design and Testing, Implementation, and Performance Analysis.

The program will read in a testing data file consisting 50 reference strings of length 30 (Testing-Data.txt) – this file will be generated by you. In this project, assume that

(1) the length of the reference string is always 30, e.g. 36172472035472014635321456701 2.

(2) there are 8 pages, from 0 to 7.

You are required to run this testing data file using the three page replacement algorithms with four different page frame sizes (3, 4, 5 and 6).

You can structure the testing data in your own way. A look of a sample testing data file, named "TestingData.txt", is given as follows:

```
[Begin of TestingData.txt]
36172472035472014635321456701 2
72630162520376726172656251625 0
...
34251436152437263543526154726 5
[End of TestingData.txt]
```

You can implement the algorithms in your choice of data structures based on the program language of your choice. Note that you always try your best to give the most efficient program for each problem.

Submission Instructions:

turn in the following @canvas.cpp.edu after the completion of all three parts, part 1, part 2, and part 3

(1) testing data file (with 50 reference strings of length 30)

(2) three program files (your choice of programming language with proper documentation)

(3) this document (complete all the answers)

Project 2

Part 1: Design & Testing (30 points)

(a) Design the program by providing pseudocode or flowchart for each page replacement algorithm.

First-In-First-Out (FIFO)

```
Input: pages = [P_1, P_2, ..., P_n], n, capacity
Output: pageFaults
begin
    create empty set S;
    create empty queue Q;
    pageFaults ← 0;
    foreach i from 1 to n do
        currentPage ← pages[i];
        if size(S) < capacity then
            if currentPage not in S then
                add currentPage to S;
                enqueue currentPage into Q;
                pageFaults ← pageFaults + 1;
            end
        end
        else
            if currentPage not in S then
                oldest = dequeue from Q;
                remove oldest from S;
                add currentPage to S;
                enqueue currentPage into Q;
                pageFaults ← pageFaults + 1;
            end
        end
    end
    return pageFaults;
end
```

Algorithm 1: First-In-First-Out (FIFO)

Least Recently Used (LRU)

```
Input: pages = [P_1, P_2, ..., P_n], n, capacity
Output: pageFaults
begin
    create empty set S;
    create empty map lastUsed;
    pageFaults ← 0;
    foreach i from 1 to n do
        currentPage ← pages[i];
        if size(S) < capacity then
            if currentPage not in S then
                add currentPage to S;
                pageFaults ← pageFaults + 1;
            end
            lastUsed[currentPage] = i;
        end
        else
            if currentPage not in S then
                lruIndex ← infinity;
                lruPage ← null;
                foreach page in S do
                    lruIndex ← lastUsed[page];
                    lruPage ← page;
                end
                remove lruPage from S;
                add currentPage to S;
                pageFaults ← pageFaults + 1;
            end
            lastUsed[currentPage] = i;
        end
    end
    return pageFaults;
end
```

Algorithm 2: Least Recently Used (LRU)

Optimal Page Replacement (OPT)

```
Input: pages = [P_1, P_2, ..., P_n], n, capacity
Output: pageFaults
Function predict(pages, frames, n, startIndex):
    farthest ← startIndex;
    res ← -1;
    foreach f from 1 to size(frames) do
        found ← false;
        foreach j from startIndex to n do
            if frames[f] = pages[j] then
                found ← true;
                if j ¿ farthest then
                    farthest ← j;
                    res ← f;
                end
                break;
            end
        end
        if found = false then
            return f;
        end
    end
    return res = -1 ? 0 : res;
begin
    create empty list frames;
    pageFaults ← 0;
    foreach i from 1 to n do
        currentPage ← pages[i];
        if currentPage is in frames then
            continue;
        end
        if size(frames) < capacity then
            append currentPage to frames;
        end
        else
            index ← predict(pages, frames, n, i+1);
            frames[index] = currentPage;
        end
        pageFaults ← pageFaults + 1;
    end
    return pageFaults;
end
```

Algorithm 3: Optimal Page Replacement (OPT)

(b) Design the program correctness testing cases. Give at least 4 testing cases (with 3, 4, 5, or 6, page frames) to test your program, and give the expected correct output (# of page faults) of the program for each case in order to test the correctness of each algorithm.

Table 1: Input Testing Cases

Testing case #	**Input string reference**	**Expected # of page faults for FIFO (✓ if Correct after testing in Part 3)**	**Expected # of page faults for LRU (✓ if Correct after testing in Part 3)**	**Expected # of page faults for Optimal Algorithm (✓ if Correct after testing in Part 3)**
1 (3 page frames)	23657134517245	14✓	14✓	13✓
2 (4 page frames)	23657134517245	11✓	13✓	12✓
3 (5 page frames)	23657134517245	8✓	8✓	10✓
4 (6 page frames)	23657134517245	8✓	8✓	9✓

(c) Design testing strategy for the programs. Discuss about how to generate and structure the randomly generated inputs for experimental study later in Part 3.

To study the performance of the three page replacement algorithms, let's use a random number generator for generating the testing data set file of 50 reference strings of length 30 as the input for the programs. However, student should use this same data set for running each of the three page replacement algorithms and each of the four frame sizes (3, 4, 5, 6).

The average performance (average number of page faults) can be calculated after an experiment is conducted using the testing data set file of 50 reference strings of length 30.

For example, if $X = \{x_1, x_2, x_3, \ldots, x_{50}\}$ contains 50 results (number of page faults) using the testing data set as input for FIFO and page frame size of 3, then average number of page faults for FIFO and page frame size of 3

$$= \frac{\sum_{i=1}^{50} x_i}{50}$$

Testing Strategy Design

We need to verify the correctness and evaluate the performance (average page faults) of each page replacement algorithm under consistent conditions. To do that, we'll create multiple controlled random inputs (reference strings) for different frame sizes (3 frames, 4 frames, 5 frames, 6 frames) and feed the same inputs to all three algorithms (FIFO, LRU, OPT).

Generate Random Inputs

1. Reference String Length: Each reference string will be randomly generated using a uniform distribution from 8 pages.

   ```
   pages = list(range(8))
   rng.choice(pages)
   str(rng.choice(pages)) for x in range(length=30)
   ```

2. Frame Sizes: 3 frames, 4 frames, 5 frames, 6 frames. This ensures consistency across algorithms and various frame sizes.
3. Data File Structure (TestingData.txt)

   ```
   1. 10433211600133036374026542351 1
   2. 61554071614531034131647525534 1
   ...
   50. 16162451154476275705640165202 7
   ```

   Each reference string is exactly 30 pages long with 8 different pages.
4. Random Reference String Generation: For each input size, generate 50 different reference strings (sequences). We'll do this using a simple random number generator in Python.

```python
import random
rng = random.Random(seed)
pages = list(range(8))
lines = []
for _ in range(num_strings):
    seq = "".join(str(rng.choice(pages)) for _ in range(length))
    lines.append(seq)
with open("TestingData.txt", "w", encoding="utf-8") as f:
    f.write("\n".join(lines) + "\n")
```

Performance Measurement

For each frame size $n \in \{3, 4, 5, 6\}$, we will run all three algorithms on the same 50 random reference strings. Then record the average number of page faults for each sequence:

$$X = \{x_1, x_2, \ldots, x_{50}\}$$

where each x_i is the number of page faults for one reference string. Lastly, compute the final performance metric:

$$\text{Average Number of Page Faults} = \frac{\sum_{i=1}^{50} x_i}{50}.$$

We can compare these average values across all three page replacement algorithms to analyze efficiency and fairness.

Part 2: Implementation (30 points)

(a) Code each program based on the design (pseudocode or flow chart) in Part 1(a).

(b) Document the program appropriately.

(c) Test you program using the designed testing input data given in the table in Part 1(b), Make sure each program generates the correct answer by marking a "✓" if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

(d) For each page replacement program, capture a screen shot of the execution (Compile & Run) of one testing reference string to show how this program works properly.

By now, three working programs are created and ready for experimental study in the next part, Part 3.

The code is well-documented and runs for each page replacement algorithm. The following image shows the output of the number of page faults for the following page reference string.

```
((myenv) ) dkhun@Devins-MacBook-Air CS4310-OS % python3 PageReplacement.py
--- Reference String ---
Sequence: 23657134517245
FIFO with frame size 3: 14 page faults
FIFO with frame size 4: 11 page faults
FIFO with frame size 5: 8 page faults
FIFO with frame size 6: 8 page faults
LRU with frame size 3: 14 page faults
LRU with frame size 4: 13 page faults
LRU with frame size 5: 8 page faults
LRU with frame size 6: 8 page faults
OPT with frame size 3: 13 page faults
OPT with frame size 4: 12 page faults
OPT with frame size 5: 10 page faults
OPT with frame size 6: 9 page faults
```

Figure 1: Output of program execution with number of page faults

Part 3: Performance Analysis (40 points)

(a) Run each program with the designed randomly generated input data given in Part 1(c). Generate a table for all the experimental results for performance analysis as follows.

Table 2: Average Number of Page Faults

number of page frames	**Average number of page faults (FIFO Program)**	**Average number of page faults (LRU Program)**	**Average number of page faults (Optimal Algorithm)**
3 page frames	19.42	19.44	19.04
4 page frames	16.32	16.30	16.00
5 page frames	13.86	13.24	13.40
6 page frames	11.04	10.74	10.98

(b) Plot a graph of each algorithm, average page fault vs. page frame size (3, 4, 5, 6) and summarize the performance of each algorithm based on its own graph.

Plot all three graphs on the same graph and compare the performance (page faults) of all three algorithms. Rank three page replacement algorithms. Try giving the reasons for the findings.

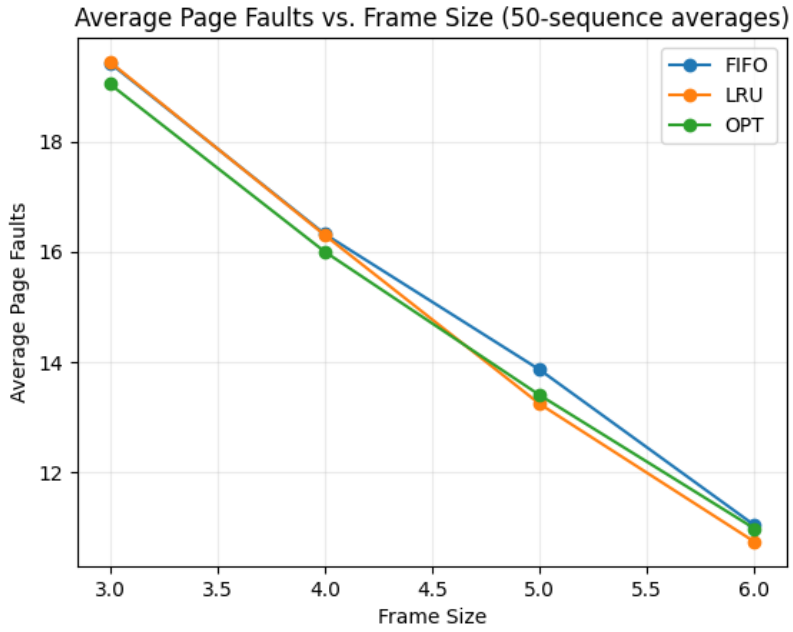
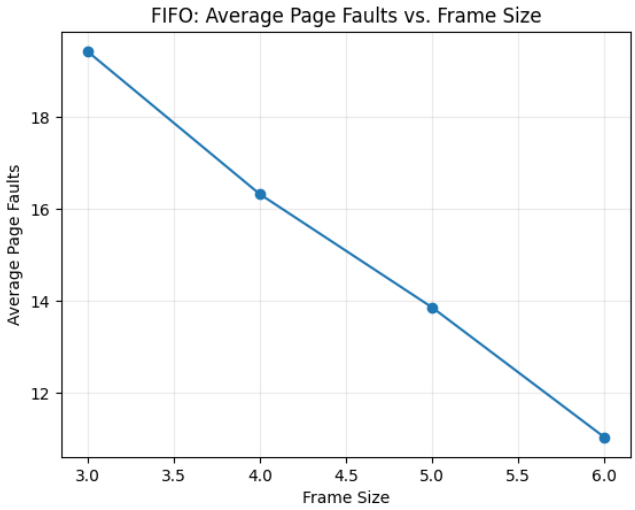
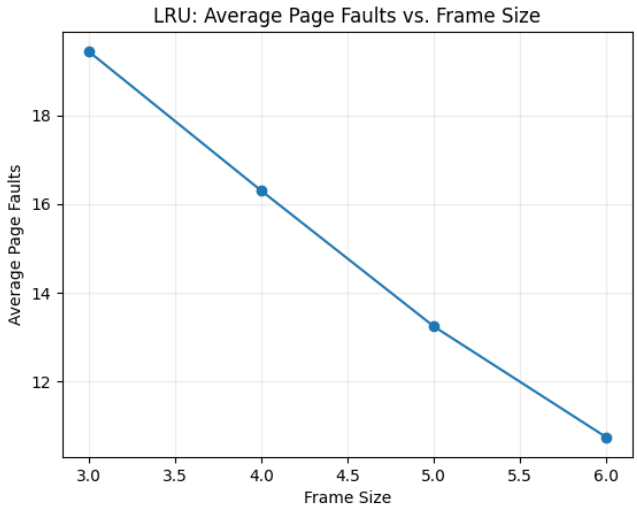


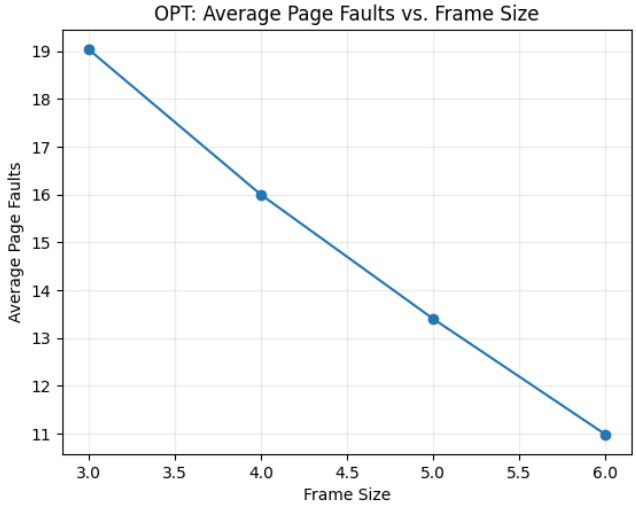
Figure 2: Average Page Faults vs. Frame Size (Combined)



(a) Average Page Faults vs. Frame Size (First-In-First-Out)



(b) Average Page Faults vs. Frame Size (Least Recently Used)



(c) Average Page Faults vs. Frame Size (Optimal Algorithm)

Figure 3: Average Page Faults vs. Frame Size for Three Page Replacement Algorithms

The overall ranking for the three page replacement algorithms based on the average page faults across different frame sizes:

1. Optimal Algorithm: 14.855
2. Least Recently Used: 14.930
3. First-In-First-Out: 15.160

Some observations I noticed in the average page faults were:

- Optimal Algorithm (OPT) is the theoretical best possible algorithm since it knows the future references. So it should always have the fewest page faults.

- Least Recently Used (LRU) is a solid real-world heuristic that tries to approximate the optimal page replacement by assuming pages not used recently won't be used soon.
- First-In-First-Out (FIFO) is the simplest and least adaptive since it ignores recency and just replaces the frams in order.

(c) Conclude your report with the strength and constraints of your work. At least 100 words. (Note: It is reflection of this project. If you have a chance to re-do this project again, what you like to keep and what you like to do differently in order get a better quality of results.)

This project helped me understand page replacement algorithms in a more practical way, especially how FIFO, LRU, and OPT behave under different memory constraints. One of the strengths of this project was that most of of the code is fully automated to test all algorithms on a predefined case and large random page sequences. This made it easy to compare behavior and verify that the implementations were accurate. The graphs and visualizations gave a clearer picture of how algorithm performance changes with frame size.

A constraint of this project is that random test data does not fully reflect real-world memory access patterns, which usually include locality of reference. Because of this, some differences between algorithms appeared smaller than expected. To improve upon this, I would include more realistic workloads and maybe even trace-driven simulations to make the analysis more meaningful. I'd also spend more time exploring edge cases that highlight where algorithms differ more sharply, and work on improving the visualizations to present the patterns in the results easier to interpret.