Sprawozdanie dotyczące symulacji porównania algorytmów planowania czasu procesora dla otwartej i zamkniętej puli zadań oraz algorytmów zastępowania stron

Autor: Kamila Szukała (272175)

Spis treści

1. Wybrane algorytmy

Dla planowania czasu procesora dla zamkniętej oraz otwartej puli zadań wykorzystałam algorytm FCFS – First Come First Serve i porównałam go z algorytmem SJF – Shortest Job First. Jeśli chodzi o planowanie priorytetowe to wybrałam algorytm FCFS priorytetowo. W przypadku zadania zastępowania stron wykorzystałam następujące algorytmy: FIFO – First In, First Out, OPT- Optimal, LRU – Least Recently Used oraz LFU – Least Frequently Used.

2. Implementacja

Do wykonania zadania wykorzystałam język programowania Python. Głównym powodem, dla którego wybrałam akurat ten język jest obszerny zestaw bibliotek, z których można skorzystać. W moim kodzie skorzystałam z kilku bibliotek. Pierwszą z nich jest 'tkinter', standardowa biblioteka Pythona służąca do tworzenia interfejsów graficznych. Wykorzystuję ją do generowania okien, przycisków, etykiet i innych elementów interfejsu. Wprowadziłam również różne moduły, takie jak 'FCFS', 'SJF', 'FCFS_with_priority', 'FIFO_algorithm', 'opt_algorithm', 'LRU_algorithm', 'LFU_algorithm'. Zawierają one różne algorytmy zarządzania procesami lub pamięcią, takie jak First-Come-First-Serve (FCFS), Shortest Job First (SJF), FCFS z priorytetami, FIFO, OPT, LRU i LFU. Stworzyłam moduły 'read_file_processes' i 'read_file_page', które służą do odczytywania informacji o procesach i stronach z plików. Biblioteka 'time' jest używana do obsługi czasu, umożliwiając na przykład wprowadzanie opóźnień w kodzie. 'Threading' to kolejna biblioteka, którą wykorzystuję do tworzenia wielowątkowych aplikacji, co umożliwia symulowanie procesów w tle. Moduł 'sys' dostarcza dostęp do niektórych zmiennych używanych lub utrzymywanych przez interpreter Pythona. Użyłam go do zamykania aplikacji. Biblioteka 'random' jest używana do generowania liczb losowych. Wykorzystałam to do stworzenia generatora nowych danych do plików tekstowych. Do wyświetlania okien dialogowych skorzystałam z 'messagebox' z biblioteki 'tkinter'. Mój interfejs graficzny (GUI) zawiera główne okno (window), w którym umieszczam różne widgety, takie jak ramki (Frame), przyciski (Button), etykiety (Label) itp. Dodatkowo, GUI posiada przycisk 'Stop', który służy do zatrzymywania symulacji procesów. Funkcja add_process otwiera nowe okno, umożliwiając użytkownikowi dodanie nowego procesu do symulacji.

3. Opis działania programu

a) Algorytm planowania czasu procesora - FCFS

W moim kodzie zastosowałam algorytm FCFS (First-Come, First-Served) – prosty sposób planowania procesów, gdzie procesy obsługiwane są według kolejności ich nadejścia. Wpierw importuję funkcję ‘read_processes’ z modułu ‘read_file_processes’, odczytując informacje o procesach z pliku. Pierwsza linijka oznacza czas przybycia procesu, druga linia – czas trwania, trzecia – priorytet, a czwarta to kwant czasu, który w przypadku planowania priorytetowego jest wykorzystywana do postarzania procesów. Kolejnym krokiem jest posortowanie procesów według czasu ich nadejścia przy użyciu funkcji ‘sorted’. Inicjalizuję zmienne przechowujące informacje o procesach, takie jak numery procesów, czasy zakończenia, czasy przetwarzania, czasy oczekiwania itp.
Następnie, dla każdego procesu z posortowanej listy, wykonuję szereg kroków:
Dodaję numer procesu do listy numerów procesów. Dodaję czas nadejścia procesu do listy czasów nadejścia. Dodaję czas wykonania procesu do listy czasów wykonania.
Obliczam czas zakończenia procesu jako sumę obecnego czasu zakończenia i czasu wykonania procesu. Jeśli czas nadejścia procesu jest większy niż obecny czas zakończenia, ustawiam czas zakończenia na czas nadejścia procesu.
Obliczam czas przetwarzania procesu jako różnicę między czasem zakończenia a czasem nadejścia.

Obliczam czas oczekiwania procesu jako różnicę między czasem przetwarzania a czasem wykonania.

Na zakończenie, obliczam średni czas przetwarzania i średni czas oczekiwania jako średnią arytmetyczną odpowiednich czasów dla wszystkich procesów. Zwracam wszystkie zebrane informacje, które mogą być wykorzystane w innych częściach programu, np. w interfejsie graficznym (GUI).

b) Algorytm planowania czasu procesora - SJF

Algorytm SJF (Shortest Job First), czyli ‘Najkrótszy Najpierw’, to strategia planowania procesów, która wybiera do wykonania proces o najkrótszym czasie wykonania spośród dostępnych procesów. W moim kodzie zastosowałam algorytm SJF, a oto jak on działa:

Rozpoczynam od importu funkcji ‘read_processes’, która służy do odczytywania informacji o procesach z pliku. Inicjalizuję zmienne, takie jak ‘t’ jako czas, ‘gantt_chart’ jako wykres Gantta, ‘completed’ jako zakończone procesy i ‘process_list’ jako listę procesów do wykonania. Wykres Gantta w przypadku algorytmu SJF służy do wizualizacji sekwencji wykonywania procesów, umożliwiając monitorowanie kolejności, czasów przetwarzania oraz identyfikację okresów oczekiwania. Jest przydatny do analizy efektywności algorytmu SJF w minimalizacji czasów wykonania i oczekiwania. Następnie, dopóki lista procesów nie jest pusta, wykonuję następujące kroki:

Tworzę listę ‘available_processes’ dostępnych procesów, które mają czas przybycia mniejszy lub równy obecnemu czasowi. Jeśli nie ma dostępnych procesów, dodaję ‘Idle’ do wykresu Gantta, zwiększam czas o 1 i kontynuuję do następnej iteracji pętli. Jeśli są dostępne procesy, sortuję je według czasu wykonania i wybieram proces o najkrótszym czasie wykonania. Dodaję identyfikator procesu do wykresu Gantta, zwiększam czas o czas wykonania procesu i obliczam czas zakończenia, czas przetwarzania i czas oczekiwania procesu. Następnie usuwam proces z listy procesów do wykonania i dodaję go do słownika zakończonych procesów.

Na zakończenie obliczam średni czas przetwarzania i średni czas oczekiwania jako średnią arytmetyczną odpowiednich czasów dla wszystkich zakończonych procesów.

Z funkcji zwracam wykres Gantta, słownik zakończonych procesów, średni czas przetwarzania i średni czas oczekiwania, aby wykorzystać je w GUI.

c) Algorytm planowania priorytetowego czasu procesora – FCFS priorytetowo

W moim kodzie zaimplementowano algorytm FCFS (First-Come, First-Served) z uwzględnieniem priorytetów. Procesy są obsługiwane na podstawie ich priorytetu, a w przypadku równej wartości priorytetów - według kolejności czasu przybycia. Jeśli w danej chwili w kolejce znajduje się tylko jeden proces, zostaje on natychmiast obsłużony. Warto zaznaczyć, że procesy w kolejce są jedynie tymi, które już przybyły. W trakcie działania kodu, inicjalizuję puste listy do przechowywania informacji o procesach. Dla każdego procesu z listy ‘priority_processes’, dodaję informacje o nim do odpowiednich list. Tworzę kopię listy ‘priority_processes’ jako ‘queue’ i słownik ‘process_dict’, który mapuje numery procesów na ich odpowiednie listy informacji.

Następnie, dla każdego procesu w kolejce (‘queue’), wykonuję kroki:

Tworzę listę dostępnych procesów (‘available_processes’) o czasie przybycia mniejszym lub równym obecnemu czasowi. Sortuję tę listę według priorytetu, a następnie czasu przybycia. Dla każdego procesu w ‘available_processes’, obliczam czas oczekiwania i aktualizuję jego priorytet na podstawie tego czasu. Jeśli są dostępne procesy, wybieram ten o najniższym priorytecie jako bieżący (‘current_process’) i usuwam go z kolejki. Obliczam czas oczekiwania i przetwarzania dla obecnego procesu, aktualizuję obecny czas (‘t’) o czas jego wykonania i dodaję go do listy zakończonych procesów (‘completed_processes’). Jeśli w kolejce znajduje się tylko jeden proces, zostaje on natychmiast obsłużony.

Dodatkowo, jeśli proces czeka już ponad zdefiniowany kwant czasu (4 linijka pliku), jego priorytet jest obniżany, co skutkuje szybszym jego obsłużeniem.

Na zakończenie obliczam średni czas oczekiwania i przetwarzania, wyświetlam te wartości, listę zakończonych procesów (‘completed_processes’), a następnie zwracam te informacje.

d) Algorytm zastępowania stron – FIFO

W implementacji algorytmu FIFO (First-In, First-Out), zastosowałam następujące kroki:

Importowałam funkcję ‘read_file’, która odczytuje ciąg odwołań do stron i liczbę dostępnych ramek z pliku. Zdefiniowałam funkcję `FIFO`, przyjmującą listę stron, liczbę stron i pojemność pamięci podręcznej jako argumenty. Inicjalizowałam listy ‘s’ i ‘indexes’ do przechowywania stron w pamięci podręcznej oraz ich indeksów. Dodatkowo, ustawiłam zmienne ‘page_faults’ i ‘page_hits’ do zliczania błędów strony i trafień strony. Dla każdej strony w liście stron, wykonywałam poniższe kroki. Jeśli strona nie była w pamięci podręcznej, dodawałam ją do pamięci podręcznej (jeśli było miejsce) lub zastępowałam najstarszą stronę tą stroną (jeśli pamięć podręczna była pełna). Zwiększałam liczbę błędów strony o 1. Jeśli strona już była w pamięci podręcznej, zwiększałam liczbę trafień strony o 1.

Na koniec zwracałam liczbę trafień strony, liczbę błędów strony i historię stanów pamięci podręcznej.

e) Algorytm zastępowania stron – OPT

W implementacji algorytmu optymalnego (OPT), zastosowałam następujące kroki:

Zdefiniowałam funkcję ‘opt_algorithm’ z dwoma argumentami: ‘pages’ (strony do odwiedzenia) i ‘frames’ (liczba dostępnych ramek w pamięci podręcznej). Zainicjowałam zmienne: “page_faults’, “page_table’, “page_table_history’ oraz “list_page_faults’. Dla każdej strony w ‘pages’, sprawdzałam, czy strona już znajduje się w ‘page_table’. Jeśli strona nie była w ‘page_table’, sprawdzałam, czy ‘page_table’ jest pełna. Jeśli nie, dodawałam daną stronę do listy. Jeśli jednak była pełna, kod szukał strony do zastąpienia. Strona do zastąpienia była tą, która będzie używana najpóźniej w przyszłości. Jeśli nie było przyszłych stron, zastępowana była pierwsza strona w ‘page_table’. Następnie bieżąca strona zastępowała tę do zastąpienia. W obu przypadkach liczba błędów strony była zwiększana o 1. Jeśli strona już była w tablicy stron, kod przechodził do następnej strony. Po przetworzeniu wszystkich stron, funkcja zwracała liczbę błędów strony, historię ‘page_table’ oraz listę błędów strony.

f) Algorytm zastępowania stron – LRU

W moim kodzie zaimplementowałam algorytm LRU (Least Recently Used), który jest strategią zarządzania pamięcią. Poniżej przedstawiam kroki, jakie podjęłam w tym celu:

Zdefiniowałam funkcję ‘LRU’ z trzema argumentami: ‘pages’ (strony do odwiedzenia), ‘n’ (liczba stron) i ‘capacity’ (liczba dostępnych ramek w pamięci podręcznej). Zainicjowałam zmienne: ‘s’, ‘indexes’, ‘page_faults’, ‘page_hits’, ‘history’ i ‘list_page_faults’. Dla każdej strony w ‘pages’, kod sprawdzał, czy strona znajduje się już w ‘s’. Jeśli strona nie była w liście, kod sprawdzał, czy jest pełna. Jeśli nie, strona była dodawana do ‘s’, a jej indeks był dodawany do ‘indexes’. Jeśli jednak ‘s’ było pełne, kod szukał strony do zastąpienia. Strona do zastąpienia była tą, która była używana najdłuższy czas temu. Następnie bieżąca strona zastępowała tę do zastąpienia. W obu przypadkach liczba błędów strony była zwiększana o 1. Jeśli strona już była w ‘s’, kod zwiększał liczbę trafień strony o 1 i aktualizował indeks strony w ‘indexes’.

Po przetworzeniu wszystkich stron, funkcja zwracała liczbę trafień strony, liczbę błędów strony, historię `s` oraz listę błędów strony.

g) Algorytm zastępowania stron – LFU

W moim kodzie został zaimplementowany algorytm LFU (Least Frequently Used), który jest strategią zarządzania pamięcią. Poniżej przedstawiam kroki, które zostały podjęte w celu realizacji tego algorytmu:

Zdefiniowałam funkcję ‘LFU’ z trzema argumentami: ‘pages’ (strony do odwiedzenia), ‘n’ (liczba stron) i ‘capacity’ (liczba dostępnych ramek w pamięci podręcznej).
Zainicjowałam zmienne: ‘s’, ‘indexes’, ‘page_counter’, ‘page_faults’, “page_hits’, “history’, ‘first_occurrence’ i ‘list_page_faults’. Dla każdej strony w ‘pages’, kod sprawdza, czy strona już znajduje się w ramce. Jeśli strona nie była w ramce, sprawdzałam, czy jest ona pełna. Jeśli nie, strona była dodawana do ramki, a jej indeks był dodawany do ‘indexes’. Jeśli jednak ramka była pełna, kod szukał strony do zastąpienia. Strona do zastąpienia była tą, która była używana najrzadziej. Następnie bieżąca strona zastępowała tę do zastąpienia. W obu przypadkach liczba błędów strony była zwiększana o 1. Jeśli strona już była w ramce, kod zwiększał liczbę trafień strony o 1 i aktualizował indeks strony w ‘indexes’. Po przetworzeniu wszystkich stron, funkcja zwracała liczbę trafień strony, liczbę błędów strony, historię wszystkich referencji oraz listę błędów strony.

h) GUI – interfejs graficzny

W oknie głównym interfejsu graficznego można wybrać sobie algorytm, który ma zostać wykonany. Można wybrać dla procesów: ‘FCFS’, ‘SJF’ i ‘FCFS priority’. Po kliknięciu w przycisk wyświetla się strona z symulacją procesów. Następnie można przejść do widoku podsumowania, gdzie znajduje się tabela razem z wszystkimi obliczonymi wartościami dla każdego procesu oraz średnia. Znajduje się też tam przycisk powrotu do menu, który przenosi do widoku głównego programu. W przypadku FCFS stworzyłam algorytm dla otwartej puli zadań. W lewym górnym rogu znajduje się przycisk możliwości dodania procesu i można wybrać sobie czas wykonywania tego procesu. Czas przyjścia dobiera się automatycznie, po skończeniu wszystkich poprzednich procesów. W menu głównym znajduje się również możliwość generowania procesów oraz stron. Najpierw opiszę generator procesów. Program pyta się użytkownika o ilość procesów i jeśli poda mniej niż 15 to na ekranie wyskakuje błąd. Następnie po kliknięciu przycisku ‘Generate’, program losuje pierwszy czas przyjścia oraz wykonywania z przedziału od 0 do 5. Następnie losuje od 0 do 1, czy ma do następnej wartości dodać czy odjąć. Na końcu do następnej wartości dodaje lub odejmuje wylosowaną wartość z przedziału [0,2]. Natomiast w przypadku generowania stron, to program po prostu losuje wartości z zakresu [0,9]. Wracając do tematu menu głównego, znajdują się tam również algorytmy zastępowania stron. Po kliknięciu w wybrany z nich pojawia się ekran z pojawiającymi się co sekundę liczbami i po sekundzie pojawia się w miejscu, do którego została przydzielona (w zależności od algorytmu).

4. Porównanie średnich czasów oczekiwania procesów FCFS oraz SJF

DANE:

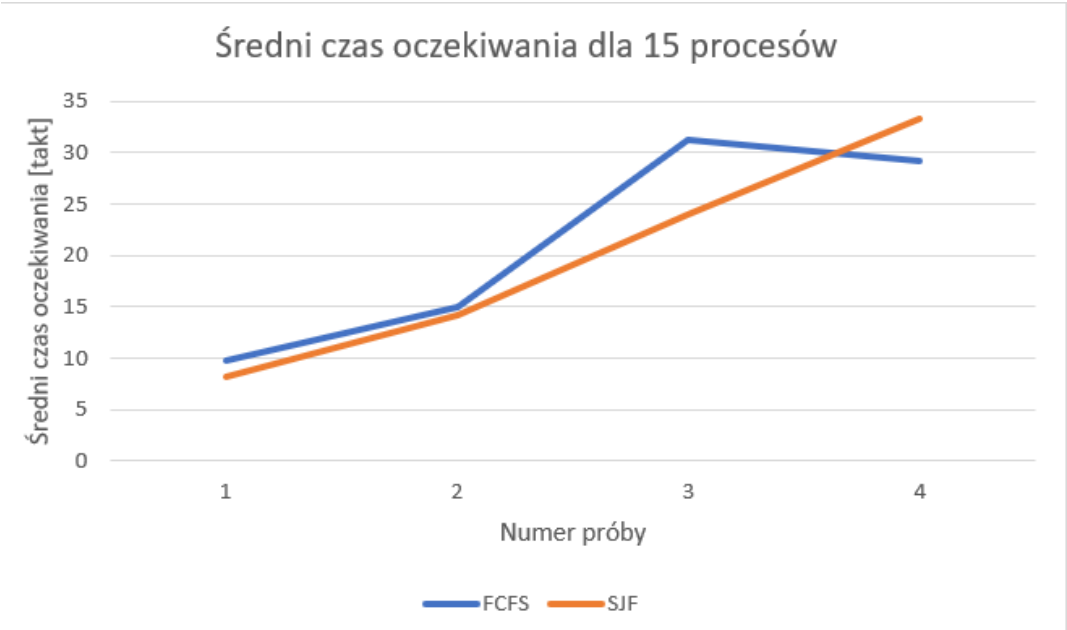
Liczba procesów: 15, 50, 100, 200.

Czas przybycia: liczba losowa nieujemna

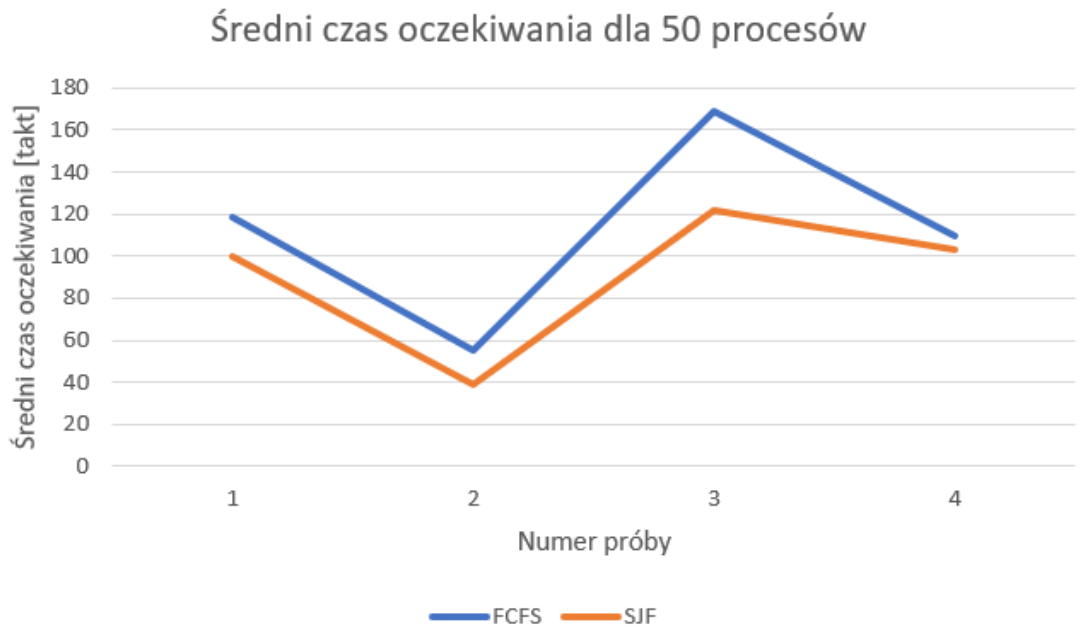
Czas wykonania: liczba losowa dodatnia

Liczba prób: 4

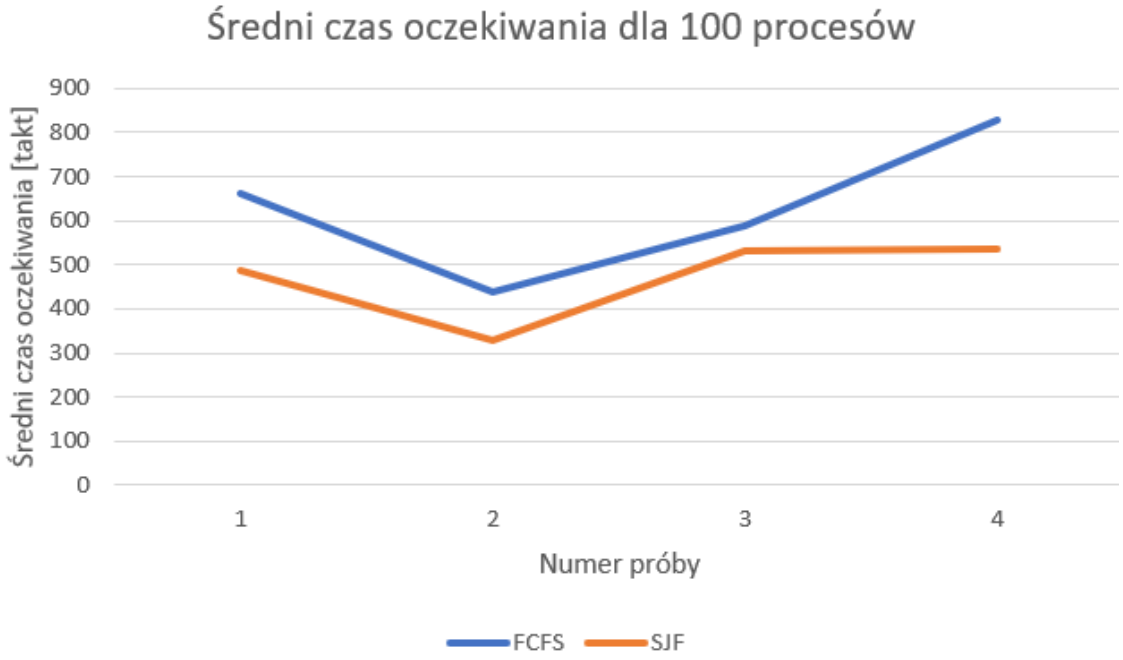
WYNIKI:



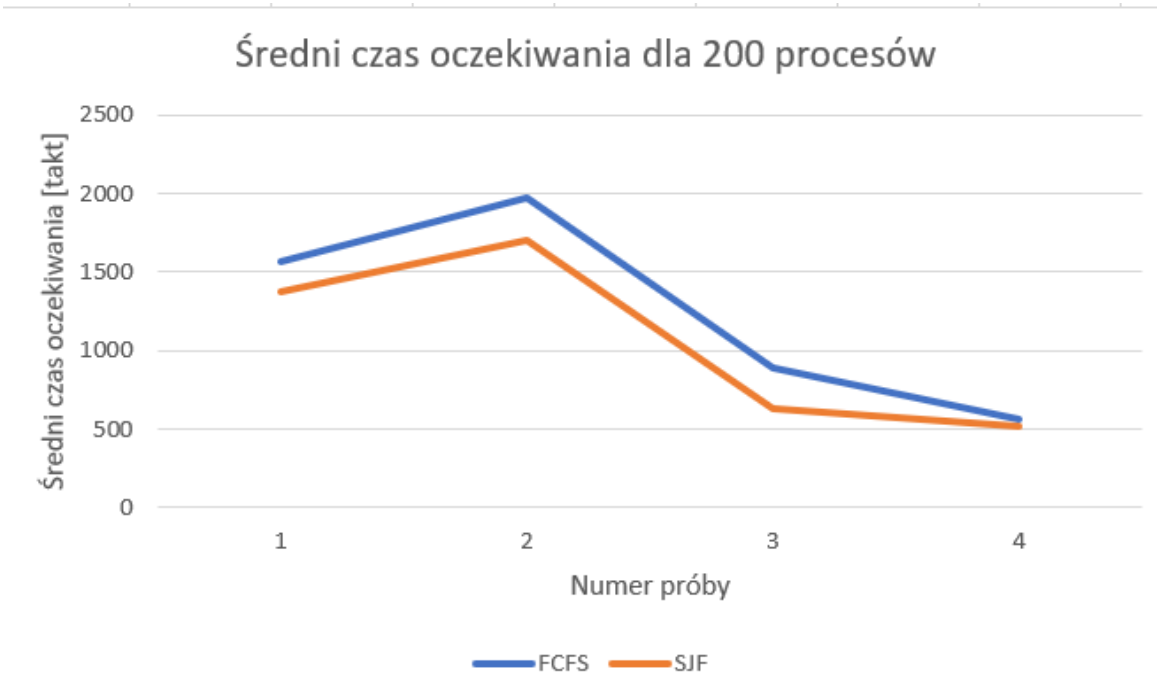
Wykres 1 – przedstawia średnie czasy oczekiwania dla 15 procesów w 4 próbach



Wykres 2 - przedstawia średnie czasy oczekiwania dla 50 procesów w 4 próbach



Wykres 3 – przedstawia średnie czasy oczekiwania dla 100 procesów w 4 próbach



Wykres 4 – przedstawia średnie czasy oczekiwania dla 200 procesów w 4 próbach

Algorytm FCFS			
Liczba procesów	50	100	200
Średni czas oczekiwania dla 5 prób	112,81	628,49	1250,06
Średni czas zwrotu dla 5 prób	125,12	639,12	1263,15
Algorytm SJF			
Liczba procesów	50	100	200
Średni czas oczekiwania dla 5 prób	90,73	470,76	1057,56
Średni czas zwrotu dla 5 prób	102,35	481,13	1068,29

Tabela 1 - przedstawia średnie arytmetyczne średnich czasów oczekiwania oraz zwrotu dla 4 prób dla poszczególnych algorytmów

WNIOSKI:

Na wstępie należy zauważyć, że długość całkowitego czasu trwania ciągu procesów nie jest zależna od algorytmu. Nieistotne jest to czy zastosujemy FCFS, czy SJF jeśli chcemy poznać czas wykonania całości. Kluczowe są za to wartości średniego czasu oczekiwania pojedynczego procesu na to aż zostanie on obsłużony i wykonany oraz średni czas zwrotu tego procesu. To na te dwie wartości wpływają algorytmy, które testowaliśmy. Patrząc na wykresy i tabelę można jednoznacznie stwierdzić, że algorytm SJF znacznie lepiej radzi sobie z optymalnym dobraniem kolejności procesów tak, aby średni czas oczekiwania oraz zwrotu była jak najniższa. FCFS w każdym przypadku potrzebował większej ilości taktów do wykonania.

5. Porównanie średnich czasów oczekiwania procesów FCFS zwykłego oraz priorytetowego

DANE:

Liczba procesów: 15, 50, 100, 200.

Czas przybycia: liczba losowa nieujemna

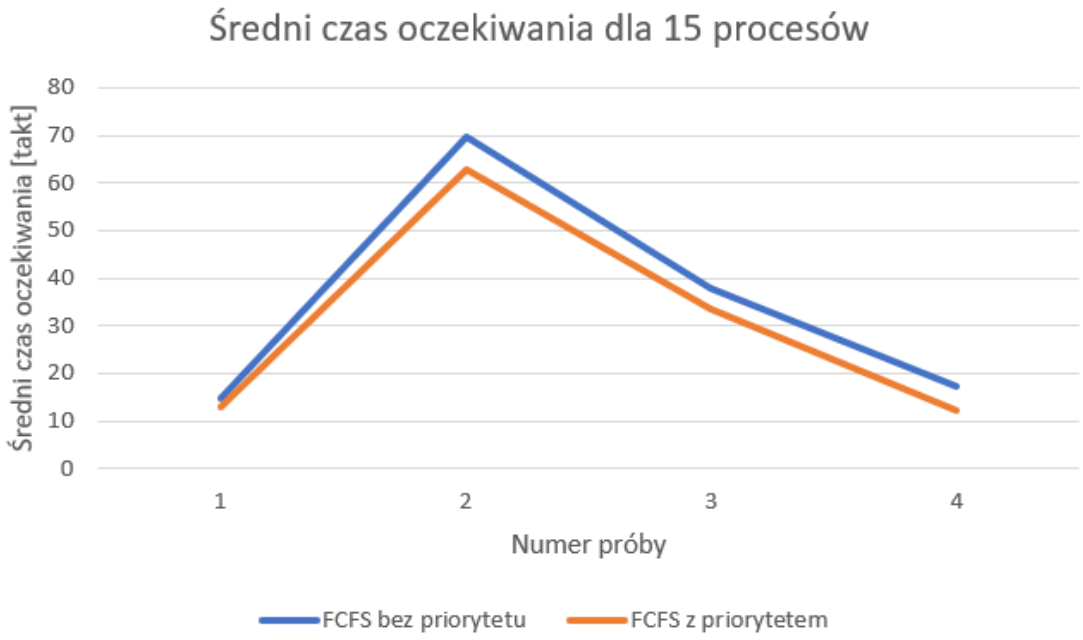
Czas wykonania: liczba losowa dodatnia

Priorytet: liczba losowa nieujemna

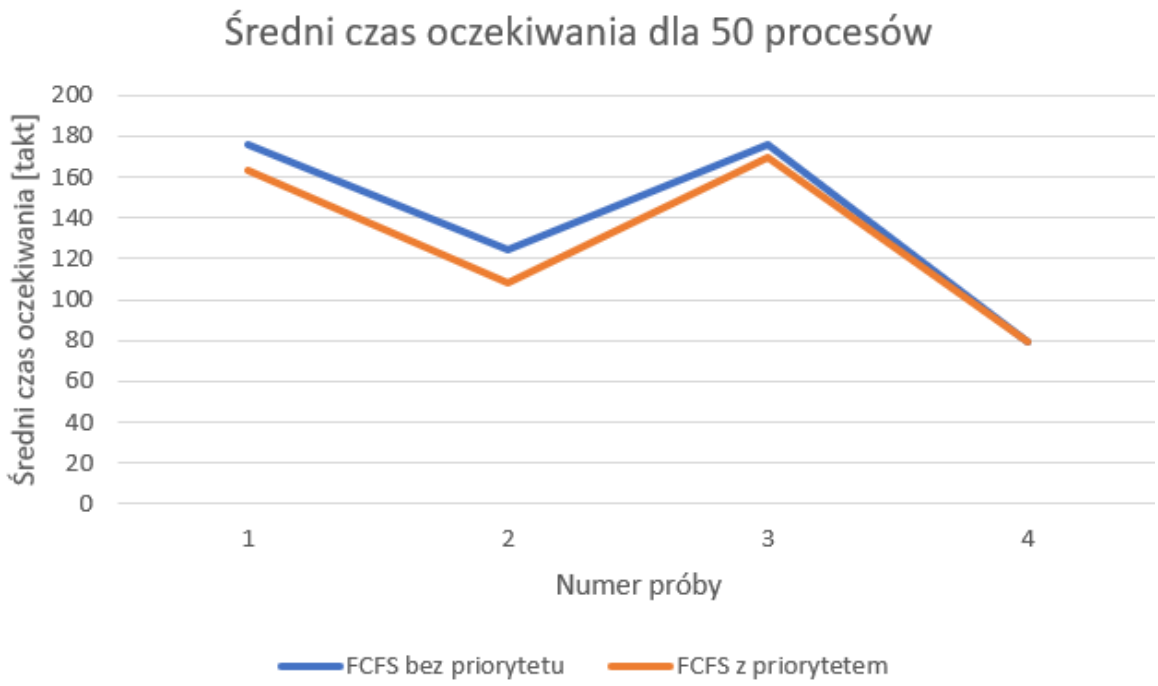
Kwant – liczba losowa nieujemna

Liczba prób: 4

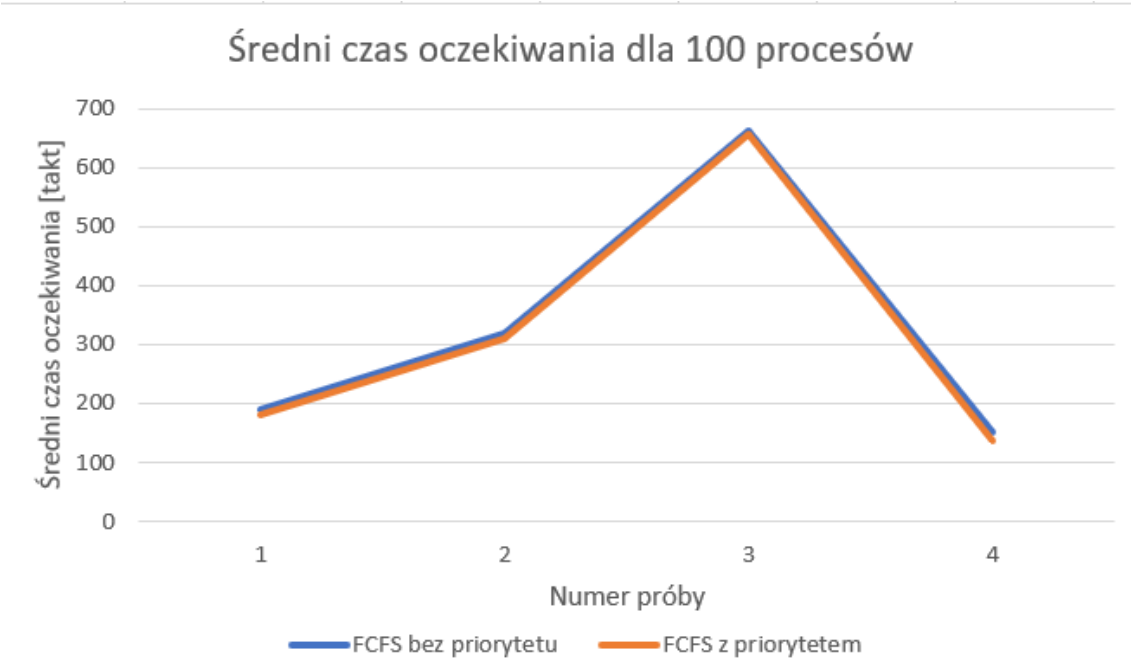
WYNIKI:



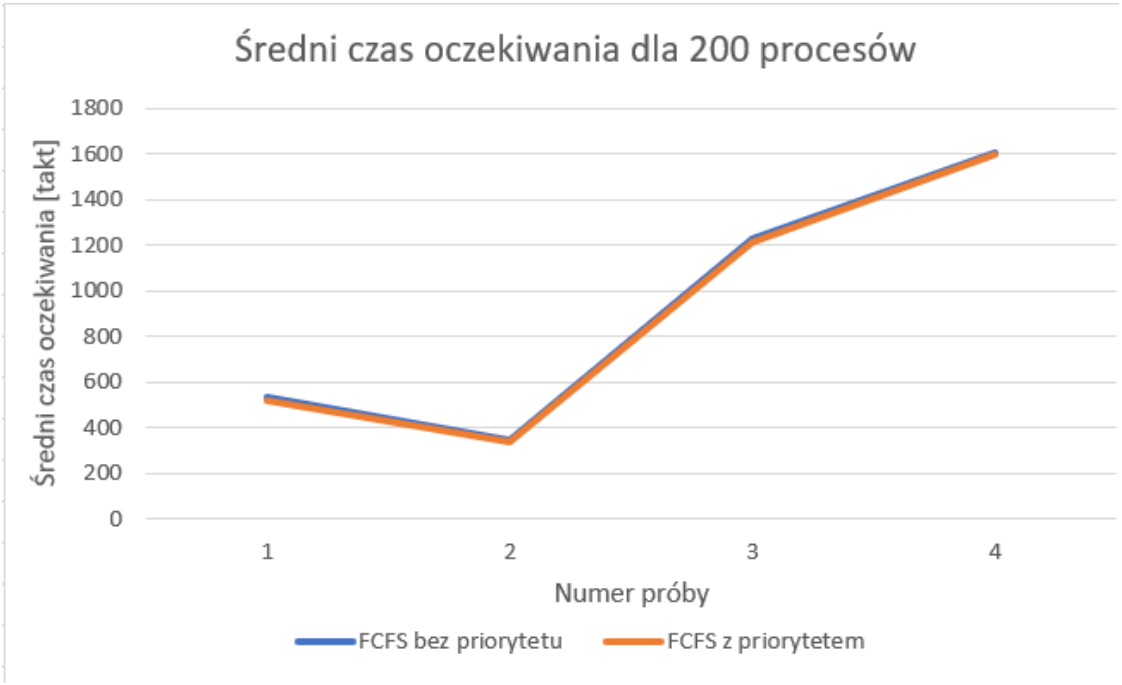
Wykres 5 - przedstawia średnie czasy oczekiwania dla 15 procesów w 4 próbach



Wykres 6 - przedstawia średnie czasy oczekiwania dla 50 procesów w 4 próbach



Wykres 7 - przedstawia średnie czasy oczekiwania dla 100 procesów w 4 próbach



Wykres 8 - przedstawia średnie czasy oczekiwania dla 200 procesów w 4 próbach

WNIOSKI:

Z powyższych wykresów można wywnioskować, że algorytm priorytetowy FCFS jest minimalnie lepszy od zwykłego, ponieważ troszkę mniej czasu zajmuje mu czas oczekiwania. Jednakże porównując wszystkie 3 algorytmy pod względem optymalności czasu oczekiwania, to wygrywa SJF.

6. Porównanie algorytmów zastępowania stron FIFO i OPT

DANE:

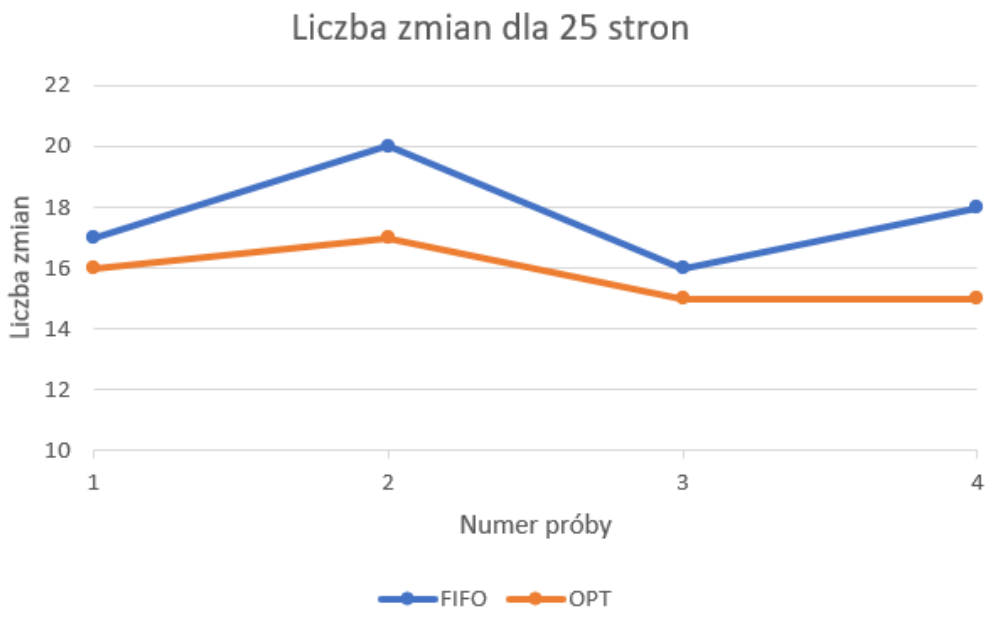
Liczba stron: 25, 50, 75, 100

Liczba różnych stron: losowa wartość z zakresu [0,9]

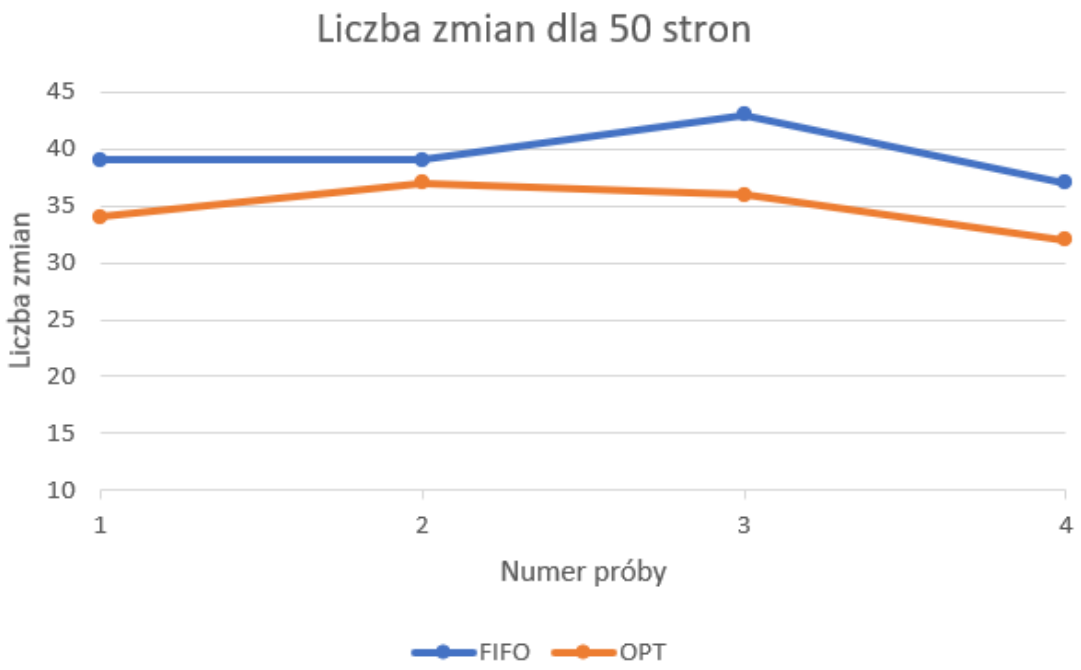
Liczba ramek: 3

Liczba prób: 4

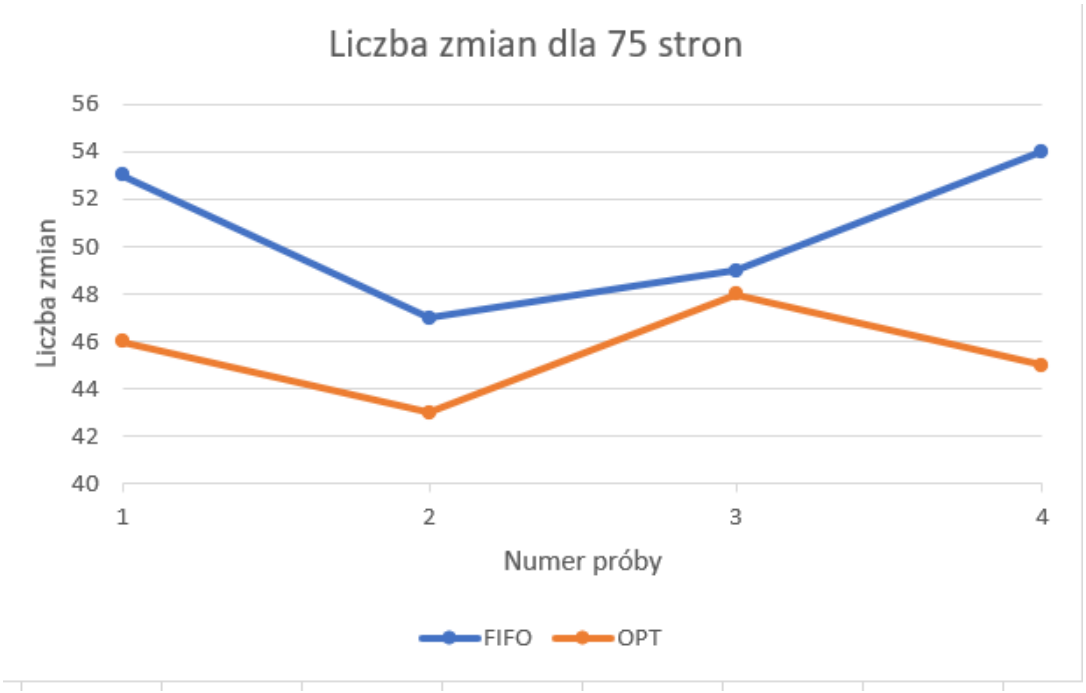
WYNIKI:



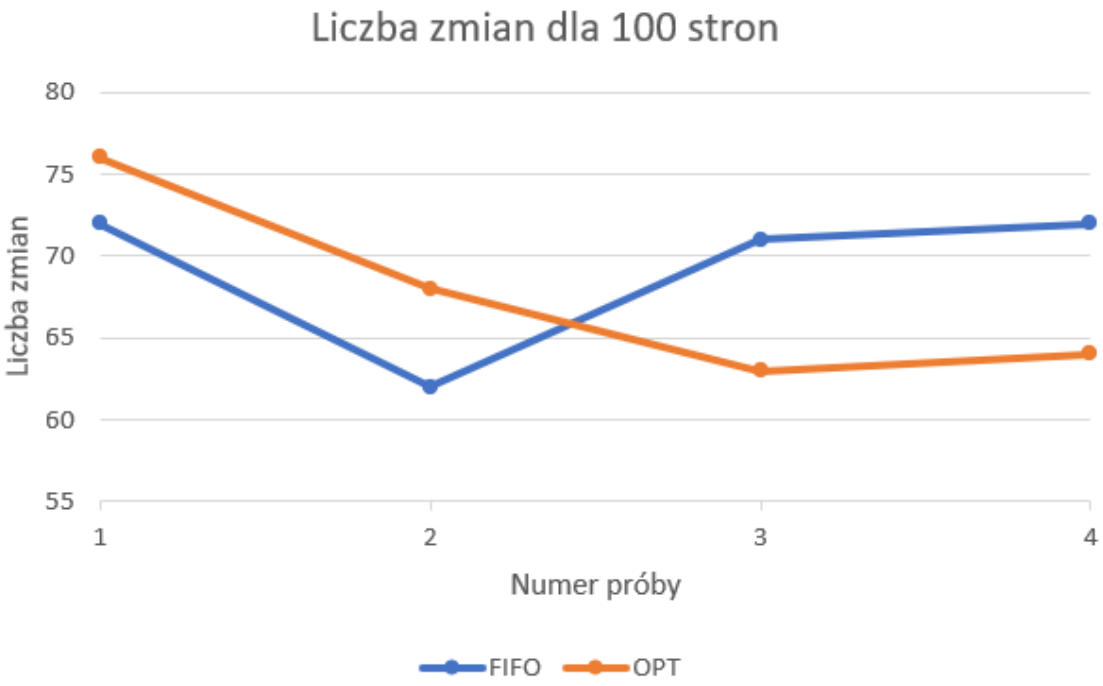
Wykres 9 - przedstawia liczbę zmian (page faults) dla 25 stron



Wykres 10 - przedstawia liczbę zmian (page faults) dla 50 stron



Wykres 11 - przedstawia liczbę zmian (page faults) dla 75 stron



Wykres 12 - przedstawia liczbę zmian (page faults) dla 100 stron

WNIOSKI:

Analizując wykresy dla 25, 50 i 75 stron, można stwierdzić, że algorytm OPT wykazuje się większą optymalnością, charakteryzując się mniejszą liczbą zmian w porównaniu do algorytmu FIFO. Chociaż zdarzają się sytuacje, gdzie FIFO osiąga lepsze wyniki dla konkretnych danych (jak obserwujemy w dwóch przypadkach na wykresie nr 12), to jednak dla większości danych algorytm OPT prezentuje się jako bardziej wydajny.

7. Porównanie algorytmów zastępowania stron LRU i LFU

DANE:

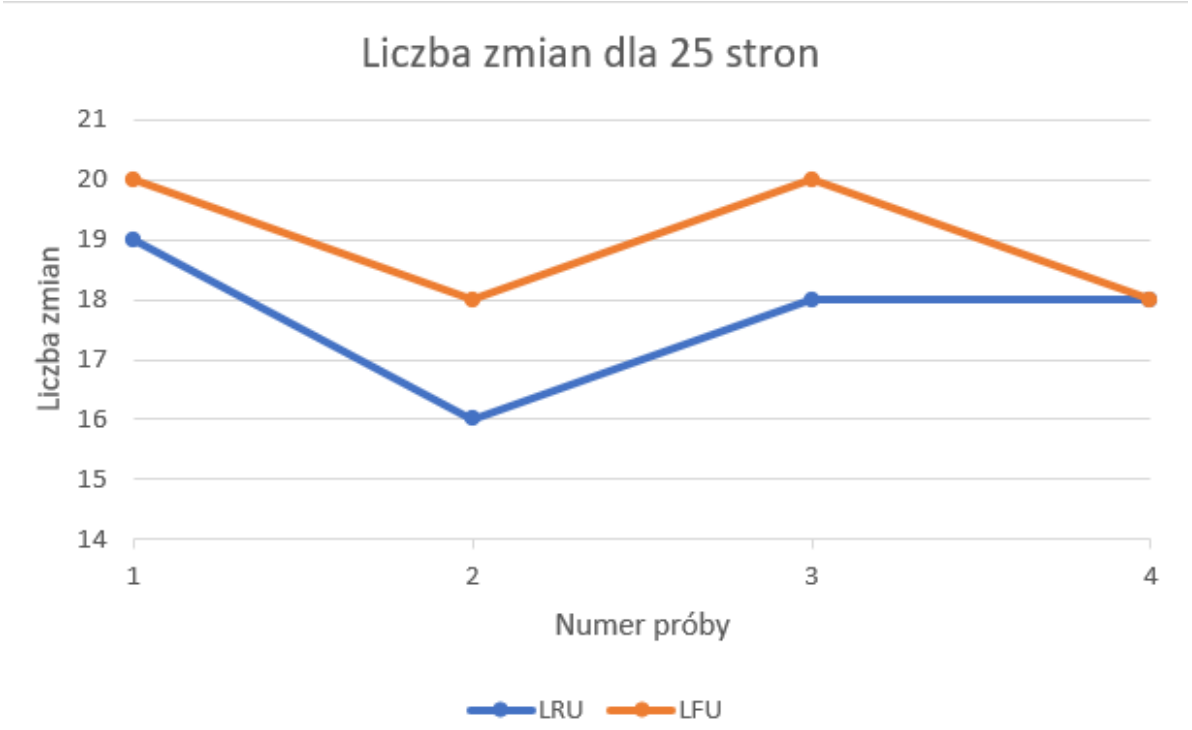
Liczba stron: 25, 50, 75, 100

Liczba różnych stron: losowa wartość z zakresu [0,9]

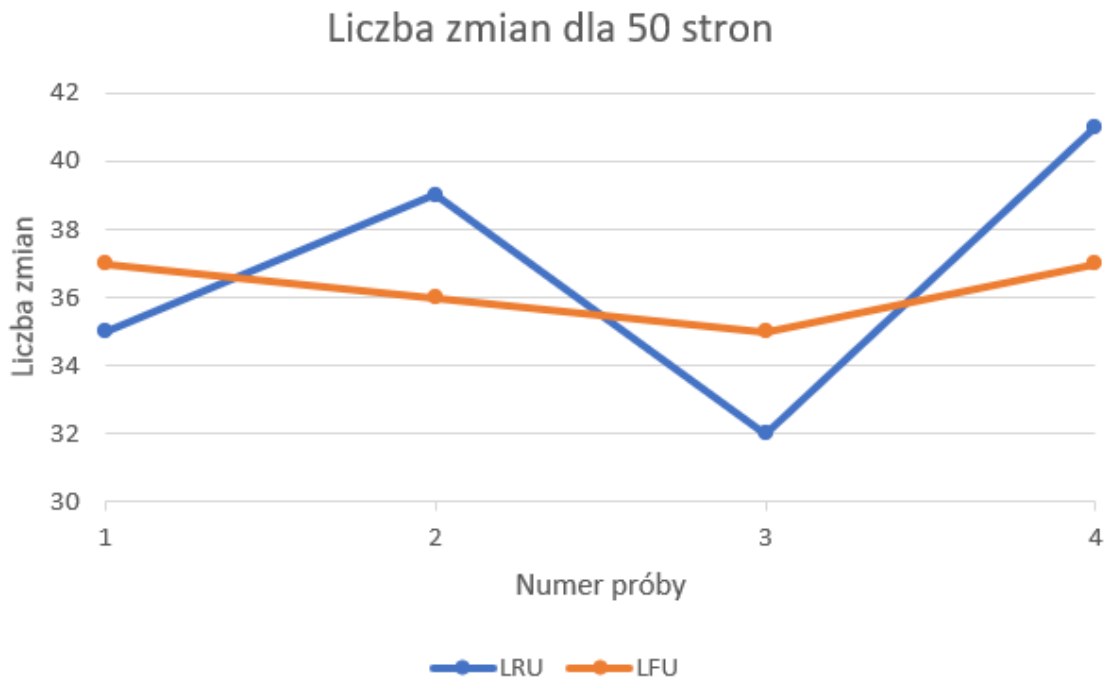
Liczba ramek: 3

Liczba prób: 4

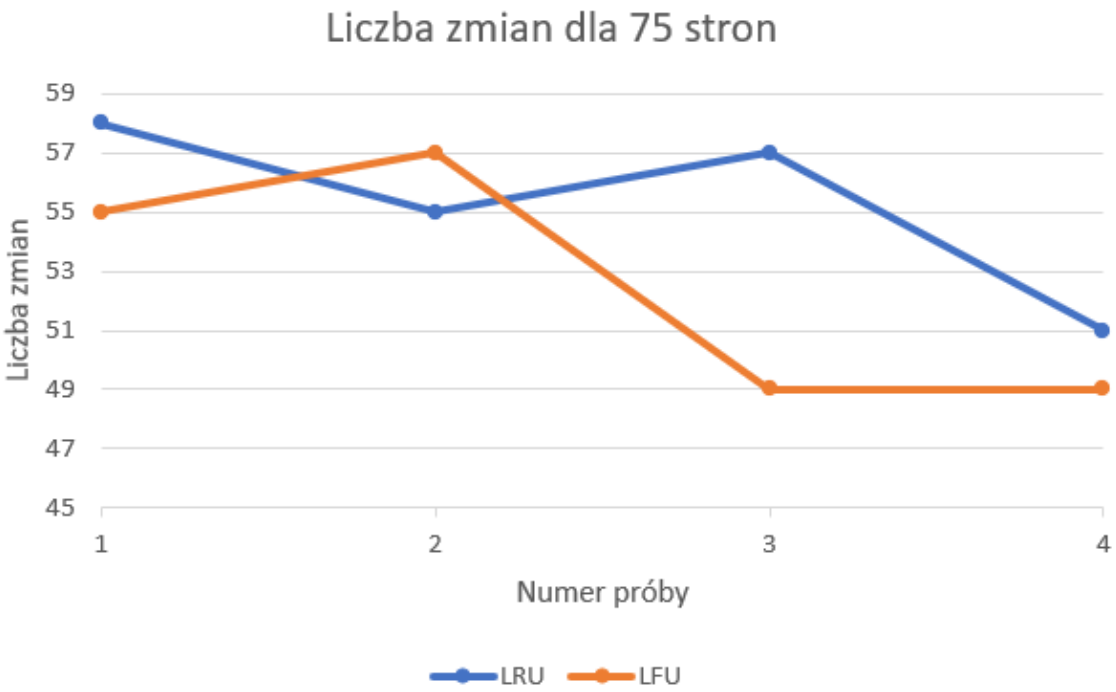
WYNIKI:



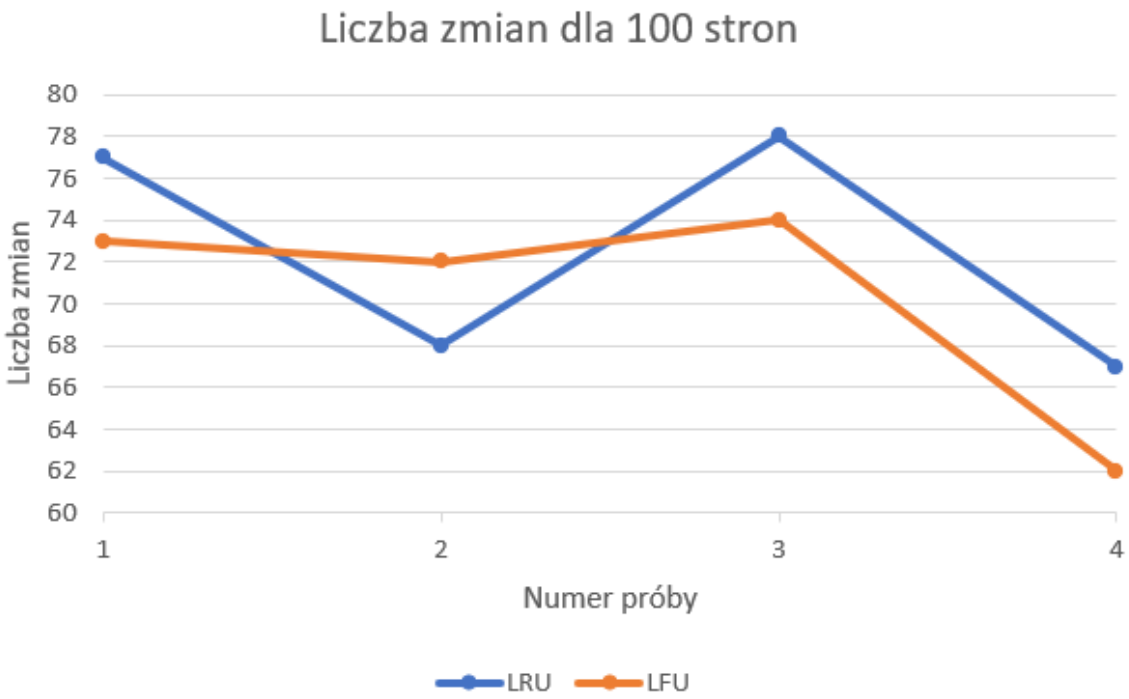
Wykres 13 - przedstawia liczbę zmian (page faults) dla 25 stron



Wykres 14 - przedstawia liczbę zmian (page faults) dla 50 stron



Wykres 15 - przedstawia liczbę zmian (page faults) dla 75 stron



Wykres 16 - przedstawia liczbę zmian (page faults) dla 100 stron

WNIOSKI:

Analizując wykresy, można zauważyć pewne podobieństwo między algorytmami LFU i LRU. Niemniej jednak, obserwując wykres numer 13 (dla 25 stron), można dostrzec, że LRU wykazuje dominację dzięki niższej liczbie zmian. Warto zaznaczyć, że im większa liczba stron, tym bardziej algorytm LFU staje się efektywny. Ogólnie rzecz biorąc, dla danych ogólnych trudno jednoznacznie określić, który algorytm jest lepszy. Jednakże, dla konkretnych przypadków danych, można już dokonać bardziej precyzyjnych ocen. Zauważalne jest, że im więcej stron, tym bardziej zaleca się skłaniać ku wykorzystaniu algorytmu LFU. W przypadku wyższej liczby stron, algorytm LFU może okazać się bardziej wydajny, ponieważ kieruje się częstością użycia, co może lepiej odzwierciedlać aktualne potrzeby systemu. Dla danych, w których pewne strony są rzadko używane, a inne są intensywnie, LFU może lepiej dostosowywać się do tych warunków, co wpływa na jego efektywność w porównaniu do LRU.

8. Porównanie algorytmów zastępowania stron FIFO, OPT, LRU i LFU.

DANE:

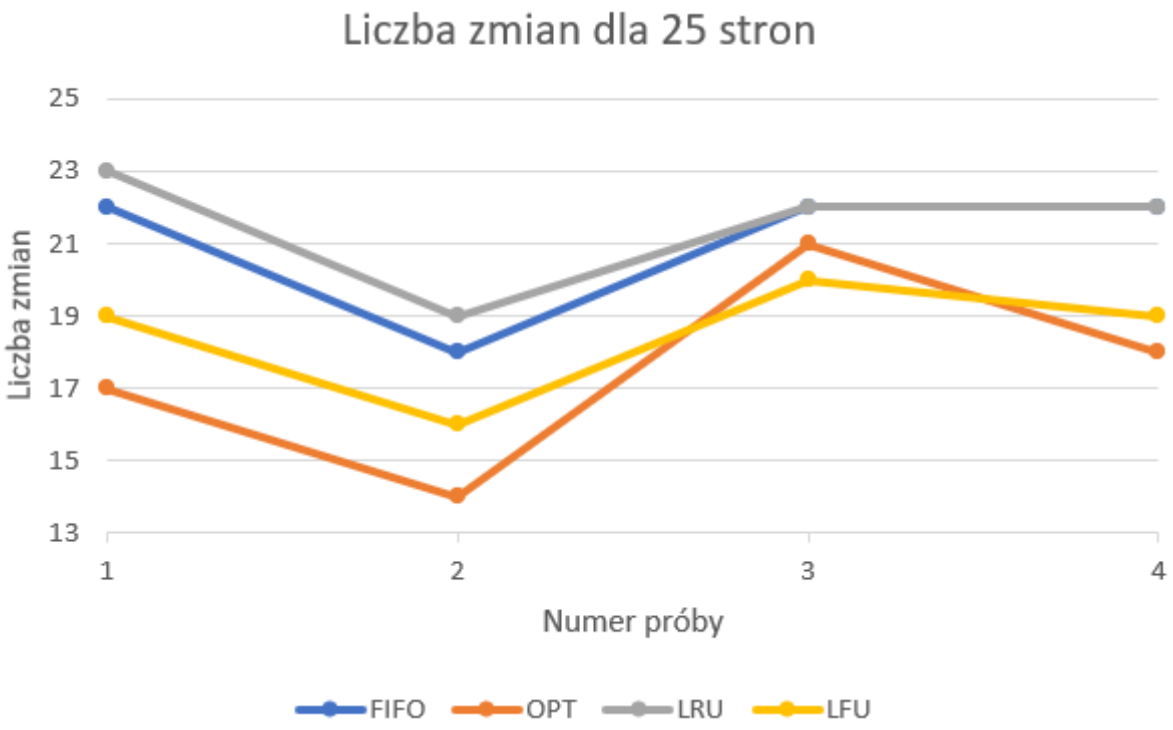
Liczba stron: 25, 100, 200, 400

Liczba różnych stron: losowa wartość z zakresu [0,9]

Liczba ramek: losowa wartość z zakresu [1,5]

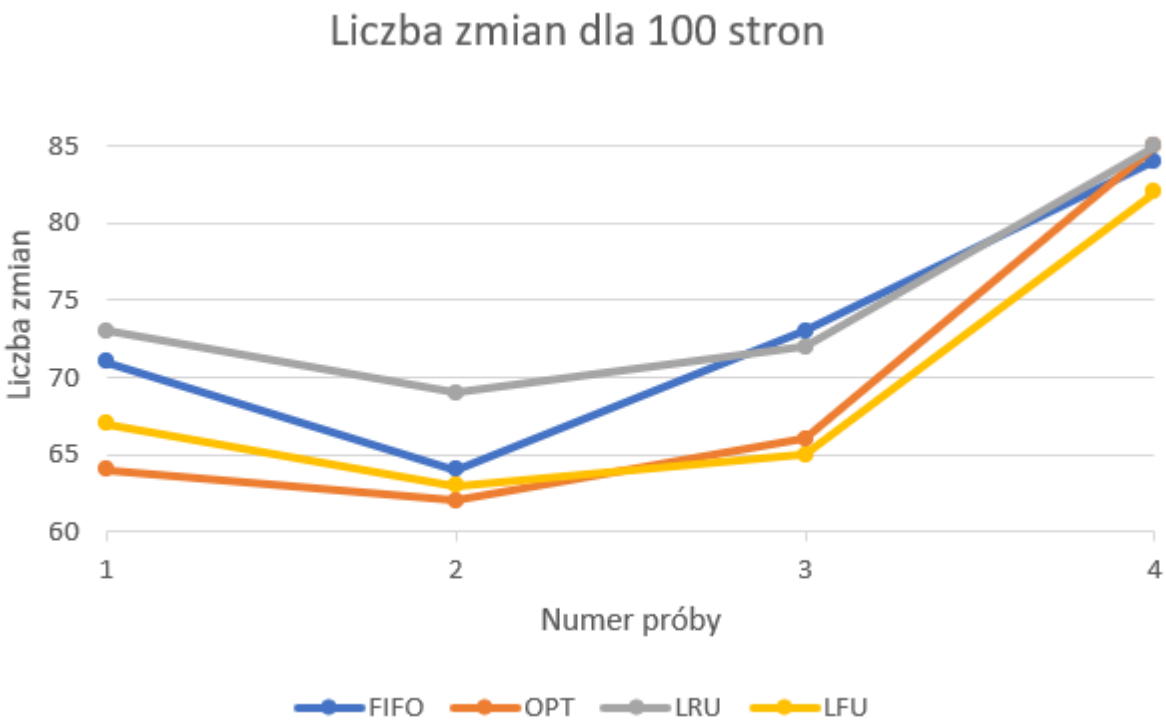
Liczba prób: 4

WYNIKI:



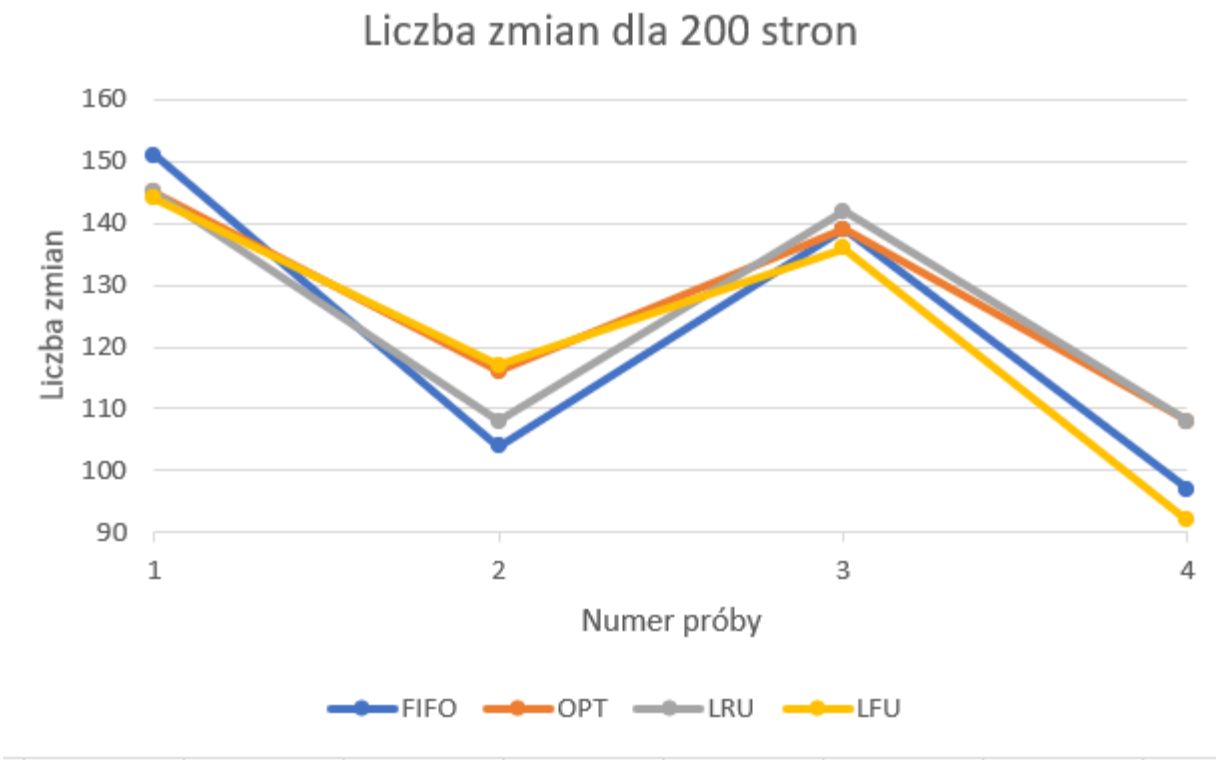
Wykres 17 - przedstawia liczbę zmian (page faults) dla 25 stron

W przypadku powyższego wykresu najbardziej wydajny jest algorytm OPT.



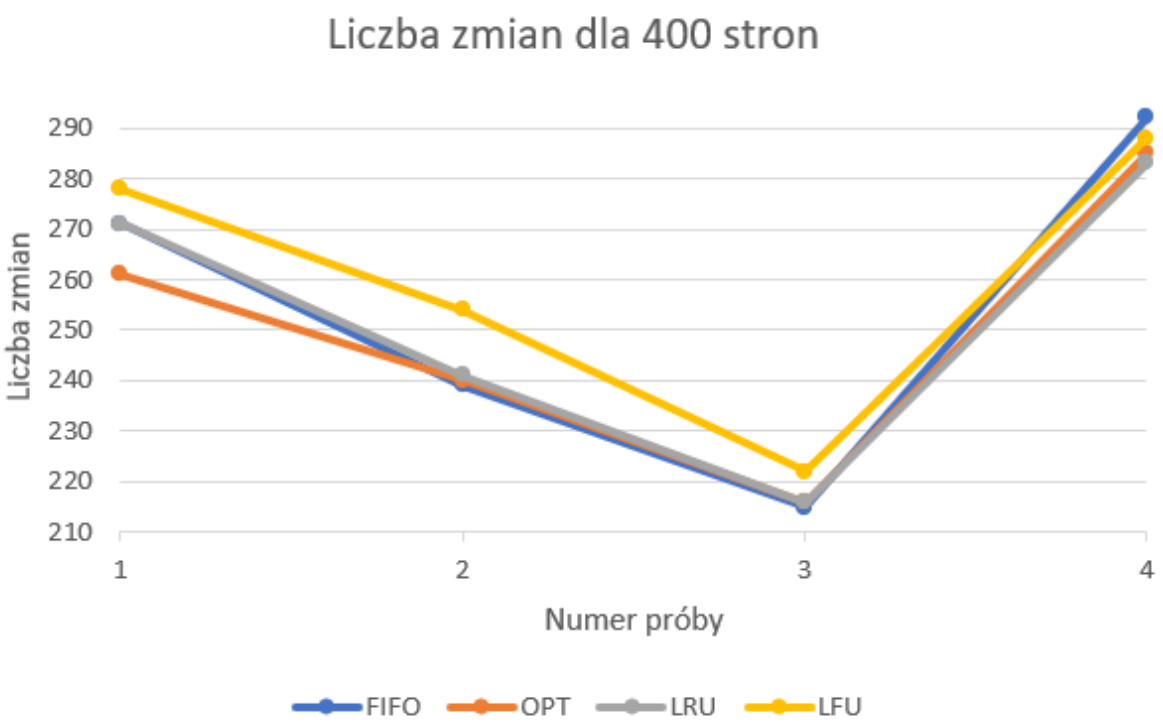
Wykres 18 - przedstawia liczbę zmian (page faults) dla 100 stron

Na powyższym wykresie można zauważyć, że najbardziej wydajne będą algorytmy OPT i LFU.



Wykres 19 - przedstawia liczbę zmian (page faults) dla 200 stron

W tym przypadku FIFO i LFU mają najmniejszą liczbę zmian.



Wykres 20 - przedstawia liczbę zmian (page faults) dla 400 stron

WNIOSKI:

Rozważając trzy pierwsze wykresy, można dostrzec, że algorytmy OPT (optymalny) i LFU (Least Frequently Used) dominują pod względem efektywności. Szczególnie zauważalne jest to na wykresie dla 25 stron, wykresie dla 100 stron oraz wykresie dla 200 stron. Algorytmy te osiągają lepsze wyniki w minimalizowaniu liczby błędów strony. Natomiast, analizując czwarty wykres związany z 400 stronami, można zauważyć, że w pierwszych trzech próbach algorytmy FIFO, OPT i LRU odgrywają kluczową rolę jako najbardziej wydajne. Jednakże, warto zauważyć, że w trakcie czwartej próby również algorytm LFU dołącza do tej elitarnej grupy algorytmów o wysokiej efektywności.

Algorytm OPT, bazujący na wiedzy o przyszłych odwołaniach, oraz algorytm LFU, eliminujący strony o najniższej częstości użycia, wykazują się skutecznością w adaptacji do różnych warunków danych. W przypadku 400 stron, mimo że trzy pierwsze algorytmy (FIFO, OPT, LRU) są początkowo najbardziej wydajne, LFU może zaczynać dominować w miarę ewolucji danych i zmian w dynamice odwołań. Ostatecznie, to zróżnicowanie podejść sprawia, że wybór optymalnej strategii zależy od charakterystyki konkretnego zestawu danych oraz ich ewolucji w czasie.

9. Podsumowanie.

a) Algorytmy planowania czasu procesora: FCFS, SJF, FCFS priorytetowo

W analizie algorytmów FCFS (First-Come, First-Served), FCFS priorytetowego (First-Come, First-Served z priorytetami) i SJF (Shortest Job First), wyniki badań wskazują, że SJF jest najbardziej efektywnym rozwiązaniem. SJF minimalizuje czas oczekiwania, eliminując zjawisko "ważenia czekania" i obsługując najkrótsze procesy jako pierwsze. W porównaniu, FCFS priorytetowy, mimo pewnej poprawy nad zwykłym FCFS dzięki priorytetom, nadal może generować niesatysfakcjonujące rezultaty, zwłaszcza w kontekście długotrwałych procesów. SJF wydaje się być bardziej uniwersalnym i efektywnym rozwiązaniem, szczególnie jeśli dysponujemy informacjami o czasie trwania procesów.

b) Algorytmy zastępowania stron: FIFO, OPT, LRU, LFU

Podczas analizy algorytmów zastępowania stron, tj. FIFO (First-In, First-Out), OPT (optymalny), LRU (Least Recently Used) i LFU (Least Frequently Used), wyniki wskazują na ich różne skuteczności. Optymalny algorytm (OPT) jest teoretycznie najbardziej efektywny, przewidując strony potrzebne w przyszłości. Jednak implementacja tego algorytmu może być trudna w praktyce, ze względu na konieczność prognozowania przyszłego używania stron. LRU i LFU to algorytmy, które stawiają na ostatnio używane lub najrzadziej używane strony. LRU skupia się na tym, które strony były używane najdawniej, podczas gdy LFU bierze pod uwagę częstotliwość użycia. FIFO jest najprostszym algorytmem zastępowania stron, działającym na zasadzie kolejki. Strony dodawane są na koniec kolejki, a najstarsza zostaje zastąpiona, gdy pamięć jest pełna. Podsumowując, optymalny algorytm ma potencjał zapewnienia najniższej liczby błędów strony, ale może być trudny do zaimplementowania. LRU i LFU oferują pewne kompromisy między złożonością a efektywnością. FIFO jest prosty, ale może prowadzić do suboptymalnych wyników w niektórych sytuacjach. Wybór algorytmu zależy od specyfiki systemu i rodzaju pracy, jaki ma wykonywać.