Sprawozdanie z implementacji i testów symulacji algorytmów planowania czasu procesora oraz zastępowania stron

Autor

Wiktoria Migasiewicz

Spis treści:

1. Implementacja oraz testy symulacji algorytmów planowania czasu procesora

1.1 Implementacja symulacji algorytmów

Swoją implementację symulacji algorytmów planowania czasu procesora w trybie niewywłaszczeniowym First Come First Served oraz Shortest Job First napisałam w języku programowania Python posługując się zintegrowanym środowiskiem PyCharm.

Na początku stworzyłam klasę *Process*, by móc tworzyć obiekty imitujące prawdziwe procesy, które obsługuje procesor. Inicjując takie obiekty, przekazuję jako argumenty czas, w którym się pojawiają - *arrival* oraz czas, który jest potrzebny na jego wykonanie – *burst*. Reszta zmiennych, potrzebnych do analizy procesów, przy tworzeniu obiektu ustawiana jest na wartość równą zeru. Dopiero w trakcie pracy procesora, mogłam posłużyć się metodą *set_finish,* przekazując jej jako argument czas, by obliczyć wartości pozostałych atrybutów. Tę klasę wykorzystałam w obydwóch symulacjach.

```
4 usages
class Process:
    def __init__(self, arrival, burst):
        self.arrival = arrival
        self.burst = burst
        self.start = 0
        self.finish = 0
        self.waiting = 0

    def set_finish(self, time):
        # When setting finish, other values are also set
        self.finish = time
        self.start = self.finish - self.burst
        self.waiting = self.start - self.arrival
```

First Come First Served

Do modułu zawierającego funkcję symulacji algorytmu First Come First Served zaimportowałam wcześniej wspomnianą klasę *Process.* Funkcja *ta* przyjmuje, jako argumenty, zbiory danych przechowujące obiekty właśnie z klasy *Process*. Te właśnie zbiory przekazuję do funkcji *len()* i tworzę zmienną *not_done*, która przechowuje liczbę niewykonanych procesów. Deklaruję zmienne *waiting_sum* oraz *time* i przypisuję im wartość równą zero, gdyż są to zmienne przechowujące sumę czasu oczekiwania wszystkich procesów oraz czas pracy procesora. Lista *queue* imituje kolejkę, w której znajdują się procesy przybyłe, lecz nie przekazane do realizacji. Zmiennymi *executing* i *burst* posługuję się, by znać atrybuty oraz stan realizacji obecnie przeprowadzanego procesu.

```
1   from processes import Process
2
3
    2 usages
4   def FCFS(processes):
5
6       not_done = len(processes)
7       waiting_sum = 0
8       time = 0
9       queue = []
10      executing = None
11      burst = None
```

Główną pętlą, która przeprowadza procesy jest pętla *while*. Dopóki każdy z przekazanych, jako argument procesów nie zostanie wykonany, pętla się wykonuje.

W swojej implementacji sprawdzam kolejno, czy w danej jednostce czasu pojawił się jakiś proces. Jeżeli procesor jest zajęty, proces zostaje dodany do kolejki, jeśli nie, zaczynamy go przeprowadzać. Jeżeli w iteracji głównej pętli *for* warunek *burst == 0* zostanie spełniony, dekrementujemy zmienną *not_done*, finalizujemy proces i aktualizujemy dane o stanie procesora oraz dane do statystyk. Jeżeli w kolejce czeka jakiś proces, zaczynamy go realizować.

Przy każdej iteracji głównej pętli zmniejszamy również zmienną *burst*, tak aby malała wraz z upływem czasu. Funkcja FCFS zwraca średnie wartości czasu oczekiwania procesów.

```
    # Simulating working processor
    while not_done > 0:

        # Checking whether any process arrived
        for process in processes:
            if process.arrival == time:

                # Adding to queue if occupied
                if executing:
                    queue.append(process)

                # If not occupied starts executing a new process
                else:
                    executing = process
                    burst = process.burst

```

```
        # Finishing a process when it's done
        if burst == 0:

            not_done -= 1
            executing.set_finish(time)
            waiting_sum += executing.waiting
            executing = None
            burst = None

            # If there are any processes in queue, starting to execute the first one
            if queue:
                executing = queue.pop(0)
                burst = executing.burst

        if burst:
            burst -= 1

        time += 1

    # Returning average time of waiting
    return waiting_sum/len(processes)
```

Shortest Job First

Moja implementacja symulacji algorytmu Shortest Job First jest podobna do wcześniej opisanej symulacji FCFS. Importuję klasę *Process* oraz w ten sam sposób deklaruję zmienne.

```
1   from processes import Process
2
3
    2 usages
4   def SJF(processes):
5
6       not_done = len(processes)
7       waiting_sum = 0
8       time = 0
9       queue = []
10      executing = None
11      burst = None
12
```

Jednak funkcja SJF, gdy napotka proces o czasie przybycia równym obecnemu czasu, niezależnie od stanu procesora, dodaje ten proces do kolejki. Wprowadziłam taką zmianę, by uwzględnić sytuację, gdy procesor rozpoczynając pracę posiada do wyboru procesy o jednakowym czasie przybycia, lecz innym czasie obsługi. Dzięki tej implementacji z obiektów Process(0, 4) oraz Process(0, 2) wybierzemy ten o czasie obsługi równym dwa. W implementacji FCFS w przypadku wystąpienia procesów o jednakowym przybyciu, obsługujemy je w kolejności, w której zostały one przekazane w liście przy wywoływaniu funkcji.

W 23 linijce kodu użyłam funkcji sortującej przy liście *queue,* imitującej kolejkę. Dzięki ustawieniu jej klucza na lambdę, wywołującą atrybut *burst* każdego procesu, jesteśmy w stanie je właśnie według tej zmiennej posegregować rosnąco.

```
    # Simulating working processor
    while not_done > 0:

        # Checking whether any process arrived
        for process in processes:
            if process.arrival == time:

                # Adding to queue if processor occupied
                queue.append(process)

                # Sorting queue by burst time
                queue.sort(key=lambda x: x.burst)
```

```
        # Starting to execute another process from queue if there are any
        if not executing and queue:
            executing = queue.pop(0)
            burst = executing.burst

        #  Finishing a process when it's done
        if burst == 0:
            not_done -= 1
            executing.set_finish(time)
            waiting_sum += executing.waiting
            executing = None
            burst = None

            # If there are any processes in queue, starting to execute the first one
            if queue:
                executing = queue.pop(0)
                burst = executing.burst

        if burst:
            burst -= 1

        time += 1

    # Returning average time of waiting
    return waiting_sum/len(processes)
```

Funkcja *SJF*, tak samo jak *FCFS*, zwraca średnią wartość czasu oczekiwania procesów, dzięki czemu jestem w stanie wyciągnąć wnioski po wywoływaniu funkcji.

1.2 Implementacja generatora

Do generowania procesów posłużyłam się zaimplementowanym przeze mnie w module *processes_generator.py.* Zaimportowałam bibliotekę *NumPy*, tak by móc generować losowe liczby z zadanych zakresów. Funkcja przyjmuje jako argumenty: liczbę procesów, która ma się znaleźć w zwracanej liście, zakres czasu przybycia oraz zakres czasu potrzebnego do obsługi.

```python
from processes import Process
import numpy as np


16 usages
def generator(num, arrival_min, arrival_max, burst_min, burst_max):

    processes = []
    burst_sum = 0

    for i in range(num):

        # Generating a random number from a given range
        arrival = np.random.randint(arrival_min, arrival_max + 1)
        burst = np.random.randint(burst_min, burst_max + 1)

        # Appending a list of processes with a new one
        processes.append(Process(arrival, burst))

    # Printing processes
    print('Processes:\n')
    for process in processes:
        print('Process with arrival: ', process.arrival, ' and burst: ', process.burst)
        burst_sum += process.burst
    print()

    # Returning a list of processes and average burst time
    return processes, burst_sum/len(processes)
```

1.3 Procedury testowania

Do realizacji przekazałam poniższe zestawy testowe:

1. Procesy z czasem nadejścia równym 0 oraz czasem realizacji od 1 do 5 w ilości 10, 50 i 100.
2. Procesy z czasem nadejścia od 0 do 10 oraz czasem realizacji równym 5 w ilości 10, 50 i 100.
3. Procesy z czasem nadejścia od 0 do 10 oraz czasem realizacji od 1 do 10 w ilości 10, 50 i 100.
4. Procesy z czasem nadejścia od 0 do 100 oraz czasem realizacji od 1 do 100 w ilości 10, 50 i 100.

Moim celem była między innymi obserwacja zachowania symulacji przy natężeniu procesów w jednym czasie. Interesowały mnie również wyniki w przypadku przekazaniu list z procesami o jednakowym czasie potrzebnym do ich realizacji, w tym przypadku wynoszącym 5 jednostek czasu. Pozostałe zestawy testowe utworzyłam by analizować zachowanie symulacji algorytmów w większym zakresie i większej losowości. Zestawy wywołałam z ilością procesów równą 10, 50 i 100 by mieć szerszy obraz. Każdy zestaw danych testowych został wywołany 5 razy, tak by móc wyliczyć średnie czasy oczekiwania.

Testy symulacji algorytmów przeprowadziłam w module *processes_tests.py.* Poniżej zamieszczam fragment kodu wywołującego algorytmy. Wyniki wywołań tych funkcji umieściłam w pliku *processes_output.pdf*.

```
from FCFS import FCFS
from SJF import SJF
from processes_generator import generator


12 usages
def printing(processes):
    print('\nAverage FCFS waiting time: ', FCFS(processes[0]))
    print('\nAverage SJF waiting time: ', SJF(processes[0]))
    print('\nAverage FCFS and SJF burst time: ', processes[1])
    return


for i in range(5):
    print('\n\nOutput nr: ', i + 1)

    # 1.1
    print("\n10 processes with arrival time 0 and burst from 1 to to 5\n")

    proc = generator( num: 10,  arrival_min: 0,  arrival_max: 0,  burst_min: 1,  burst_max: 5)
    printing(proc)
```

1.4 Opracowanie i analiza wyników testów symulacji algorytmów

Poniżej przedstawiam w tabelach i wykresach dane zgromadzone w dokumencie

processes_output.pdf.

Processes	Arrival time	Burst time	FCFS waiting time	SJF waiting time	Average burst time
10			16,50	10,90	3,30
50	0	1-5	76,64	53,86	2,96
100			151,22	118,23	3,16
10			17,50	17,50	5,00
50	0-10	5	117,92	117,92	5,00
100			242,77	242,77	5,00
10			20,60	13,30	5,70
50	0-10	1-10	140,52	97,10	5,86
100			274,21	187,19	5,49
10			228,30	131,60	54,00
50	0-100	1-100	1235,30	712,36	47,70
100			2315,37	1543,79	51,46

Tabela 1.1 Pierwsze wywołanie

Processes	Arrival time	Burst time	FCFS waiting time	SJF waiting time	Average burst time
10			10,6	10,4	3,2
50	0	1-5	74,84	53,16	2,96
100			140,67	103,53	2,89
10			17,5	17,5	5
50	0-10	5	117,22	117,22	5
100			242,83	242,83	5
10			23	14,6	5,4
50	0-10	1-10	115,94	89,44	5,64
100			276	195,59	5,66
10			218,8	119	50,6
50	0-100	1-100	1065,72	715,8	46,32
100			2359,06	1508,28	48,57

Tabela 1.2 Drugie wywołanie

Processes	Arrival time	Burst time	FCFS waiting time	SJF waiting time	Average burst time
10			17,3	10,1	3,1
50	0	1-5	75,46	53,92	3,02
100			156,58	119,46	3,18
10			16,9	16,9	5
50	0-10	5	117,22	117,22	5
100			242,5	242,5	5
10			24	16,9	6,1
50	0-10	1-10	146,94	111,4	6,12
100			268,98	183,02	5,42
10			181	112,2	47,5
50	0-100	1-100	1206,88	661,82	45,48
100			1951,66	1215,09	42,15

Tabela 1.3 Trzecie wywołanie

Processes	Arrival time	Burst time	FCFS waiting time	SJF waiting time	Average burst time
10			17,6	12,8	3,7
50	0	1-5	74,64	51,88	2,82
100			154,7	108,83	3,04
10			18,5	18,5	5
50	0-10	5	117,48	117,48	5
100			242,47	242,47	5
10			27,9	20,6	6,9
50	0-10	1-10	146,28	108,7	6,18
100			239,66	161,56	5
10			275,9	206,1	64,1
50	0-100	1-100	1201,82	742,42	46,78
100			2487,5	1852,1	54,28

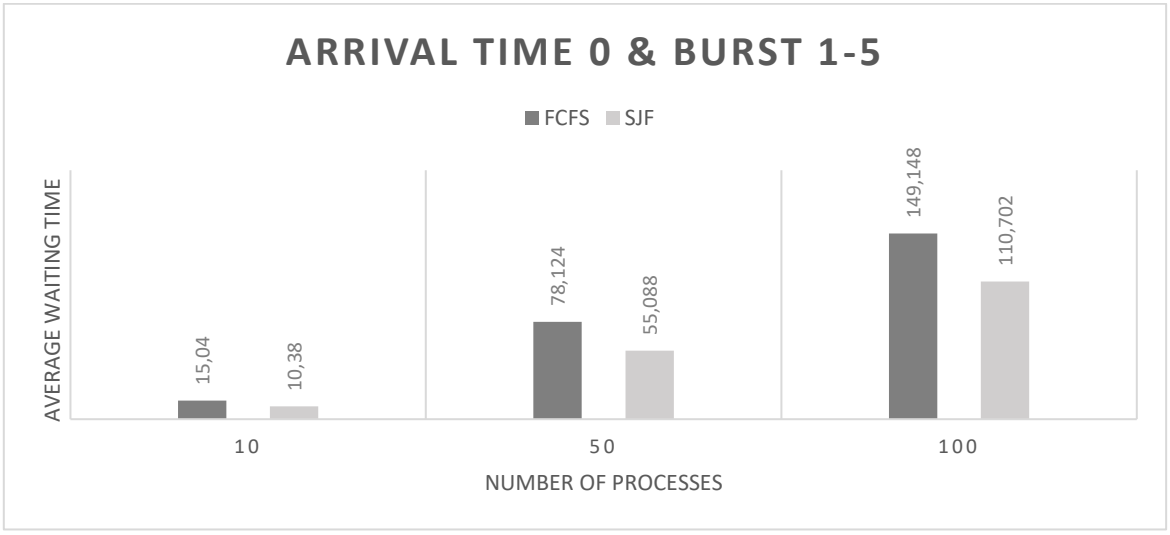
Tabela 1.4 Czwarte wywołanie

Processes	Arrival time	Burst time	FCFS waiting time	SJF waiting time	Average burst time
10			13,2	7,7	2,7
50	0	1-5	89,04	62,62	3,36
100			142,57	103,46	2,9
10			16,8	16,8	5
50	0-10	5	117,42	117,42	5
100			242,64	242,64	5
10			26,6	19,5	6,5
50	0-10	1-10	133,28	91,66	5,68
100			251,55	167,1	5,22
10			264,3	138,8	50,7
50	0-100	1-100	1432,48	1067,52	58,86
100			2452,47	1517,16	47,77

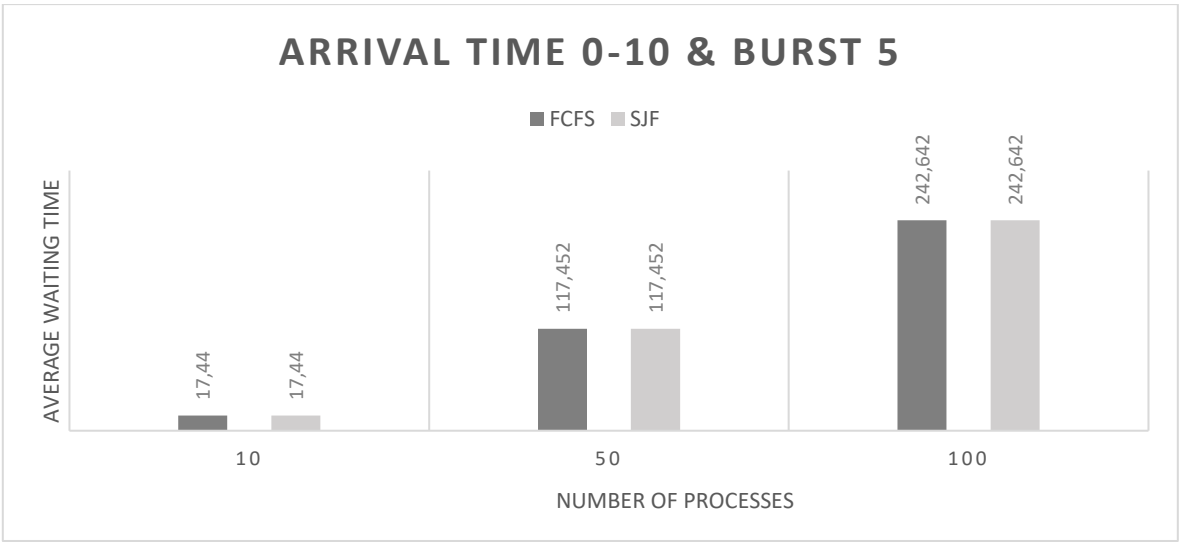
Tabela 1.5 Piąte wywołanie

Processes	Arrival time	Burst time	Average FCFS waiting time	Average SJF waiting time	Average burst time
10			15,04	10,38	3,2
50	0	1-5	78,124	55,088	3,024
100			149,148	110,702	3,034
10			17,44	17,44	5
50	0-10	5	117,452	117,452	5
100			242,642	242,642	5
10			24,42	16,98	6,12
50	0-10	1-10	136,592	99,66	5,896
100			262,08	178,892	5,358
10			233,66	141,54	53,38
50	0-100	1-100	1228,44	779,984	49,028
100			2313,212	1527,284	48,846

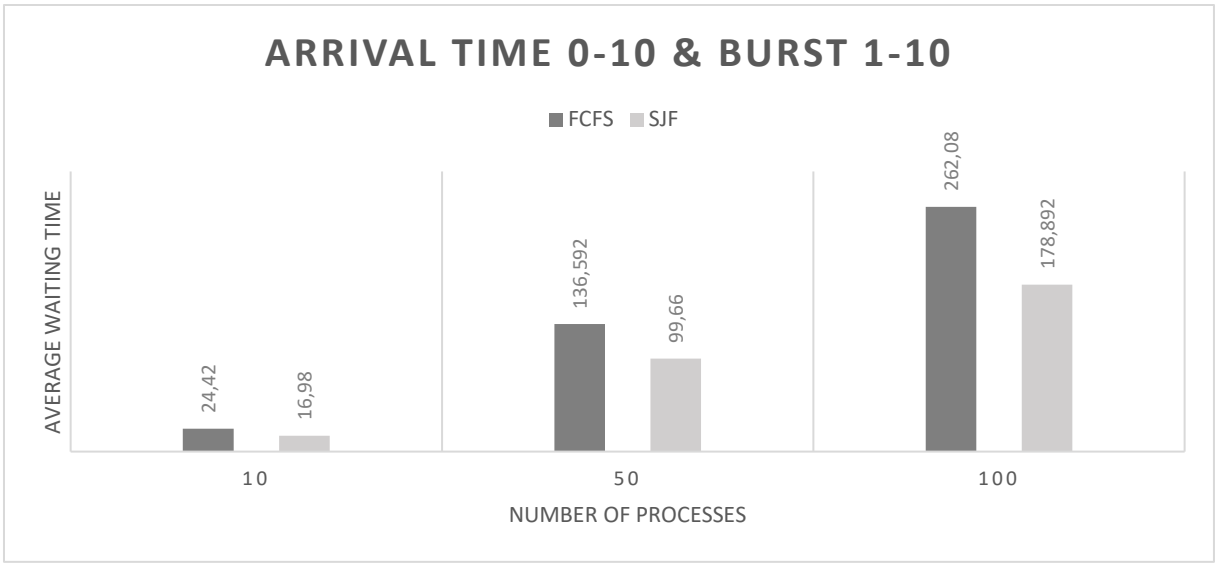
Tabela 1.6 Średnie wartości



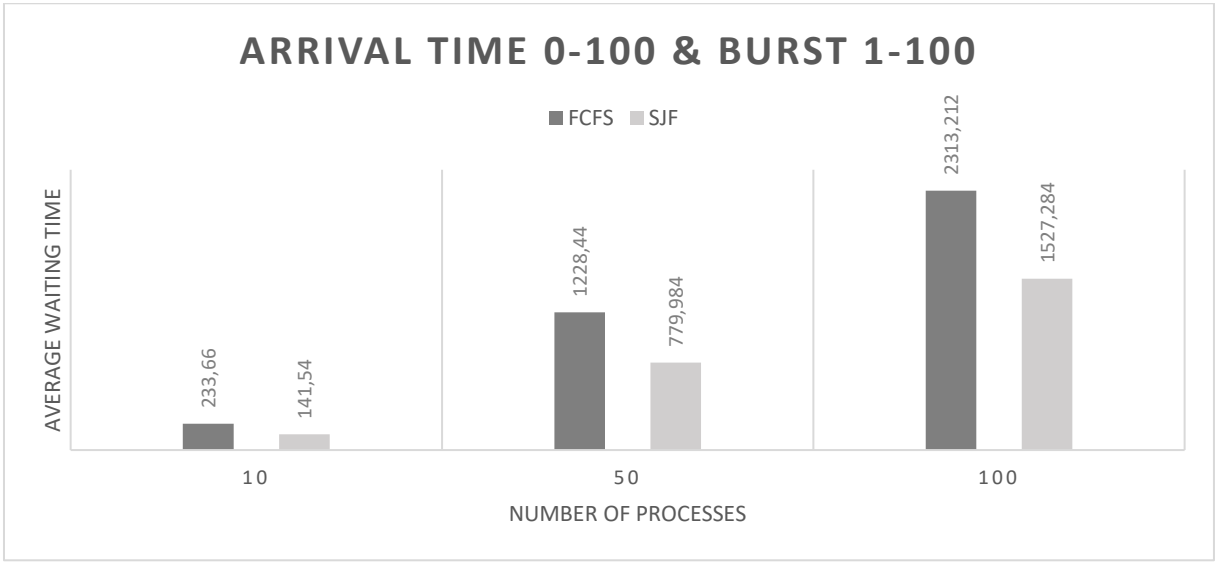
Wykres 1.1 Czas nadejścia od 0 oraz czas realizacji od 1 do 5



Wykres 1.2 Czas nadejścia od 0 do 10 oraz czas realizacji 5



Wykres 1.3 Czas nadejścia od 0 do 10 oraz burst od 1 do 10



Wykres 1.4 Czas nadejścia od 0 do 100 oraz burst od 1 do 100

1.5 Wnioski z eksperymentów

Dla wszystkich zestawów procesów FCFS generuje dość wysokie czasy oczekiwania. Średni czas oczekiwania rośnie znacząco w przypadku długich procesów, co może prowadzić do niskiej efektywności. W porównaniu do FCFS, SJF generuje w każdym testowanym przypadku znacznie niższe czasy oczekiwania, co wpływa pozytywnie na wydajność systemu. SJF, nawet dla długich procesów, utrzymuje stosunkowo dobre wyniki. Algorytm ten skutecznie minimalizuje czas oczekiwania poprzez priorytetyzację krótszych procesów. Wyniki nie różnią się, w przypadku wywołania symulacji z procesami o jednakowym czasie realizacji wynoszącym 5. Dzieje się tak, gdyż sortowanie procesów po czasie potrzebnym do ich realizacji, zastosowane w algorytmie SJF, nie wnosi żadnych zmian w kolejce, a procesy są wykonywane w jednakowy sposób. Niezależnie od zastosowanego algorytmu, czas oczekiwania wzrasta wraz z ilością procesów do wykonania. W przypadku różnych zakresów przyjścia, SJF nadal utrzymuje przewagę w skróceniu czasów oczekiwania w porównaniu do FCFS. Jest to szczególnie zauważalne dla procesów o dłuższych czasach trwania. Wybór algorytmu planowania zależy od specyfiki systemu i charakterystyki procesów. SJF zdaje się być bardziej wszechstronnym wyborem, zwłaszcza w kontekście optymalizacji czasów oczekiwania dla różnych długości procesów.

2. Algorytmy zastępowania stron

2.1 Implementacja symulacji algorytmów

Symulacje algorytmów zastępowania stron, tak jak algorytmów planowania czasu procesora, napisałam w języku programowania Python posługując się zintegrowanym środowiskiem PyCharm. By zaimplementować symulacje, stworzyłam klasę *Frame*, dzięki której mogę tworzyć obiekty reprezentujące ramki pamięci. Atrybuty takich obiektów to *index*, by móc je od siebie odróżnić, *page*, czyli strona, jaką obecnie ramka przechowuje oraz change, informująca kiedy nastąpiła w nim zmiana.

Metodą *set_change* posługuję się w symulacji algorytmu Last Recently Used, jest to mój czasowy znacznik, który pozwala mi wybrać odpowiednią ramkę pamięci w razie wystąpienia błędu strony.

```
4 usages
class Frame:
    def __init__(self, index):
        self.index = index
        self.page = None
        self.change = None

    # Setting a page and time of change
    2 usages (2 dynamic)
    def set_page(self, new_page, time):
        self.page = new_page
        self.change = time

    # Setting only time for when page already in frame
    1 usage (1 dynamic)
    def set_change(self, time):
        self.change = time
```

First In First Out

Do modułu zawierającego funkcję symulacji algorytmu First In First Out zaimportowałam wcześniej opisaną klasę z modułu *frame.py*. W implementacji tej symulacji zaczynam od zadeklarowania zmiennej *page_error*, która wyznacza ilość wystąpienia w wywołaniu braków stron oraz *time*, która służy do inkrementacji jednostki czasu.

Jako argumenty funkcja przyjmuje ilość ramek pamięci oraz zbiory danych przechowujące odwołania do stron.

```
1  from frame import Frame
2
3
   15 usages
4  def FIFO(frames, pages):
5
6      page_error = 0
7      time = 0
8
```

Pierwsza pętla *for* iteruje po każdym odwołaniu do stron z listy przekazanej funkcji, jako argument. Zmienna *page_found* jest wyznacznikiem, czy strona znajduje się już w którejś ramce pamięci, a jej wartość jest resetowana po każdym obiegu głównej pętli. Przy każdej iteracji sprawdzam między innymi dwa warunki:

1. Czy strona znajduje się już w którejś ramce pamięci?
2. Czy któraś z ramek jest pusta?

Jeśli żaden z tych warunków nie zostanie spełniony, przy czym gdy przypisujemy stronę do pustej ramki inkrementujemy zmienną *page_error* o jeden, wybieramy najodpowiedniejszą ramkę względem stosowanego algorytmu. W swojej implementacji posługuję się zmiennymi *current* i *chosen*, by umieścić stronę w ramce pamięci, w której strona pojawiła się na początku. Przy czym po takiej operacji również inkrementujemy zmienną wskazującą na ilość wystąpień błędów stron.

```
    for page in pages:
        page_found = False

        # When page is already in a frame
        for frame in frames:
            if frame.page == page:
                page_found = True
                break

        if page_found:
            time += 1
            continue

        # Checking if any frame is empty
        for frame in frames:
            if frame.page is None:
                frame.set_page(page, time)
                page_found = True
                page_error += 1
                break

        if page_found:
            time += 1
            continue

        current = None
        frame_chosen = None

        # Choosing a first-in frame
        for frame in frames:
            if current is None or frame.change < current:
                current = frame.change
                frame_chosen = frame

        # Changing page for the first-in page
        frame_chosen.set_page(page, time)
        page_error += 1
        time += 1
```

Funkcja zwraca zmienną *integer* o wartości równej ilości wystąpień błędów stron.

```

    # Returning number of page errors
    return page_error

```

Last Recently Used

Implementacja symulacji algorytmu LRU jest niemal identyczna, jak implementacja FIFO.

W ten sam sposób inicjuje zmienne, przeprowadzam proces wyboru i zwracam wartość wystąpień błędów stron. Natomiast przy pozytywnym zweryfikowaniu, czy strona znajduje się już w którejś ramce pamięci, za pomocą metody *set_change()* w 16. linijce kodu, uaktualniam czas ostatniego użycia strony. Dzięki temu w ostatniej pętli *for* wybieramy ramkę nie ‘first-in’, lecz ‘last recently used’.

```
1  from frame import Frame
2
3
   1 usage
4  def LRU(frames, pages):
5
6      page_error = 0
7      time = 0
8
```

```
9      for page in pages:
10         page_found = False
11
12         # When page is already in a frame
13         for frame in frames:
14             if frame.page == page:
15                 page_found = True
16                 frame.set_change(time)
17                 break
18
19         if page_found:
20             time += 1
21             continue
22
23         # Checking if any frame is empty
24         for frame in frames:
25             if frame.page is None:
26                 frame.set_page(page, time)
27                 page_found = True
28                 page_error += 1
29                 break
30
31         if page_found:
32             time += 1
33             continue
```

```
        current = None
        frame_chosen = None

        # Choosing a first-in frame
        for frame in frames:
            if current is None or frame.change < current:
                current = frame.change
                frame_chosen = frame

        # Changing page for the last recently used page
        frame_chosen.set_page(page, time)
        page_error += 1
        time += 1

```

Funkcja LRU również zwraca liczbę wystąpień błędów stron.

```
    # Returning number of page errors
    return page_error

```

2.2 Implementacja generatorów

W module *pages_generator.py* utworzyłam dwa generatory, pierwszy z nich to *ref_generator*, generujący listę referencji, o przekazanej jako argument, długości do stron z zakresu przekazanego, jako drugi argument. Przy pomocy biblioteki *numpy* losuję z zadanego zakresu stron liczbę całkowitą i dodaję ją do listy. Dodatkowo drukuję na konsole utworzony zbiór danych, który jest również przez generator zwracany.

Funkcja frames_generator, tworzy i zwraca zbiór obiektów typu *Frame* w ilości, przekazanej jako argument, liczbie.

```python
from frame import Frame
import numpy as np


6 usages
def ref_generator(references_num, pages_num):

    references = []

    # Creating a list of references of pages
    for i in range(references_num):
        references.append(np.random.randint( low: 0, pages_num))

    print('\nReferences:\n', references, '\n')

    return references


18 usages
def frames_generator(frames_num):

    frames = []

    # Creating frames of memory
    for i in range(frames_num):
        frames.append(Frame(i))

    return frames
```

2.3 Procedury testowania

Do realizacji przekazałam poniższe zestawy testowe:

1. 15 referencji do 10 różnych stron przy 3, 4 oraz 5 ramkach pamięci
2. 50 referencji do 10 różnych stron 3, 4 oraz 5 ramkach pamięci
3. 100 referencji do 10 różnych stron 3, 4 oraz 5 ramkach pamięci
4. 500 referencji do 500 różnych stron 10, 20 oraz 30 ramkach pamięci
5. 1000 referencji do 50 różnych stron 10, 20 oraz 30 ramkach pamięci
6. 10000 referencji do 50 różnych stron 10, 20 oraz 30 ramkach pamięci

Moim celem była między innymi analiza zachowania symulacji podczas wzrostu liczby ramek pamięci. Przetestowałam różne kombinacje. Zestawy wywołałam z ilością procesów równą 3, 4, 5 oraz 10, 20, 30, by móc obserwować zachowanie podczas dłuższej pracy.

Każdy zestaw danych testowych został wywołany 5 razy, tak by móc wyliczyć średnie ilości wystąpień błędów stron.

Testy symulacji algorytmów przeprowadziłam w module *pages_tests.py.* Poniżej zamieszczam fragment kodu wywołującego algorytmy. Wyniki wywołań tych funkcji umieściłam w pliku *output_pages.pdf*.

```python
import pages_generator
import LRU
import FIFO
from frame import Frame


18 usages
def printing(num, pag):

    frames_LRU = pages_generator.frames_generator(num)
    frames_FIFO = pages_generator.frames_generator(num)
    print('\nLRU page errors:', LRU.LRU(frames_LRU, pag), '\n')
    print('FIFO page errors:', FIFO.FIFO(frames_FIFO, pag), '\n')
    return


for i in range(5):

    print('\n\nOutput nr: ', i + 1)

    # 1.1
    print("\n3 frames of memory and 15 references to 10 pages")

    frames_num = 3
    pages = pages_generator.ref_generator( references_num: 15,  pages_num: 10)
    printing(frames_num, pages)

    # 1.2
    print("\n4 frames of memory and 15 references to 10 pages")
```

2.4 Opracowanie i analiza wyników testów symulacji algorytmów

Poniżej przedstawiam w tabelach i wykresach dane zgromadzone w dokumencie *pages_output.pdf.*

Frames	References	Pages	LRU errors	FIFO errors
3			13	14
4	15	10	11	11
5			10	10
3			38	36
4	50	10	33	33
5			29	29
3			76	74
4	100	10	65	62
5			53	56
10			409	404
20	500	50	293	283
30			188	179
10			785	791
20	1000	50	597	600
30			408	420
10			7995	7982
20	10000	50	5978	5988
30			3951	3907

Tabela 2.1 Pierwsze wywołanie

Frames	References	Pages	LRU errors	FIFO errors
3			10	10
4	15	10	10	10
5			9	10
3			36	39
4	50	10	31	29
5			24	24
3			76	78
4	100	10	63	63
5			57	53
10			403	405
20	500	50	312	306
30			226	218
10			832	827
20	1000	50	625	618
30			412	422
10			8036	8027
20	10000	50	5974	5967
30			3993	4028

Tabela 2.2 Drugie wywołanie

Frames	References	Pages	LRU errors	FIFO errors
3			13	13
4	15	10	11	11
5			10	10
3			36	35
4	50	10	33	34
5			28	31
3			74	72
4	100	10	62	62
5			53	53
10			412	416
20	500	50	306	317
30			209	222
10			809	808
20	1000	50	590	609
30			411	389
10			7983	7950
20	10000	50	6004	5957
30			3983	3980

Tabela 2.3 Trzecie wywołanie

Frames	References	Pages	LRU errors	FIFO errors
3			11	**10**
4	15	10	9	**11**
5			5	5
3			41	39
4	50	10	37	36
5			33	34
3			67	66
4	100	10	57	60
5			47	47
10			406	406
20	500	50	308	298
30			204	208
10			810	815
20	1000	50	597	590
30			403	419
10			8012	8015
20	10000	50	6047	6040
30			3987	3898

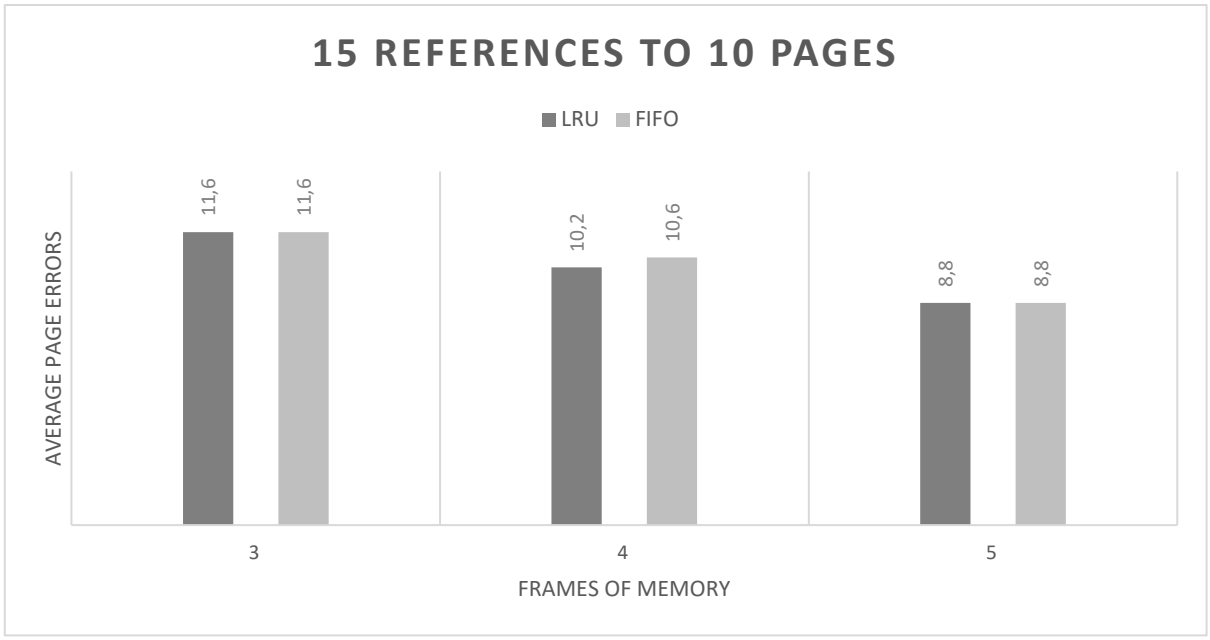
Tabela 2.4 Czwarte wywołanie

Frames	References	Pages	LRU errors	FIFO errors
3			11	11
4	15	10	10	10
5			10	9
3			36	37
4	50	10	33	33
5			30	27
3			68	72
4	100	10	62	63
5			54	54
10			418	411
20	500	50	320	321
30			224	239
10			802	791
20	1000	50	581	573
30			375	401
10			8013	8003
20	10000	50	5920	5993
30			4024	3990

Tabela 2.5 Piąte wywołanie

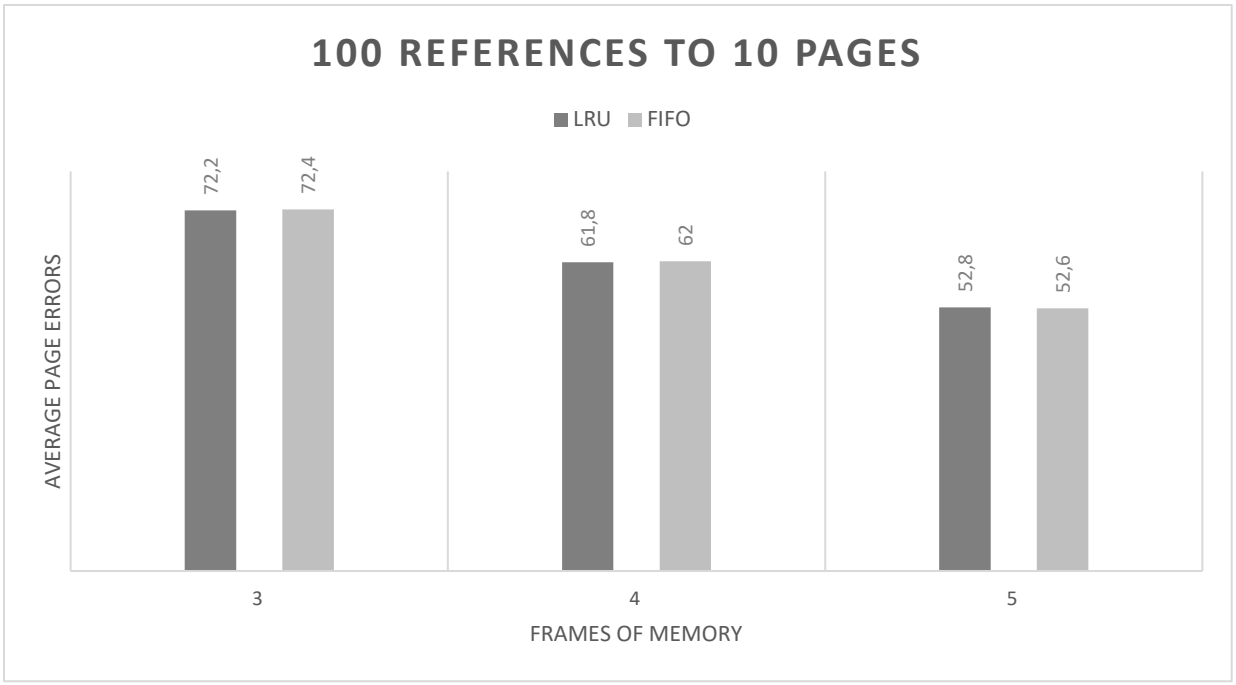
Frames	References	Pages	Average LRU errors	Average FIFO errors
3			11,6	11,6
4	15	10	10,2	10,6
5			8,8	8,8
3			37,4	37,2
4	50	10	33,4	33
5			28,8	29
3			72,2	72,4
4	100	10	61,8	62
5			52,8	52,6
10			409,6	408,4
20	500	50	307,8	305
30			210,2	213,2
10			807,6	806,4
20	1000	50	598	598
30			401,8	410,2
10			8007,8	7995,4
20	10000	50	5984,6	5989
30			3987,6	3960,6

Tabela 2.6 Średnie wartości

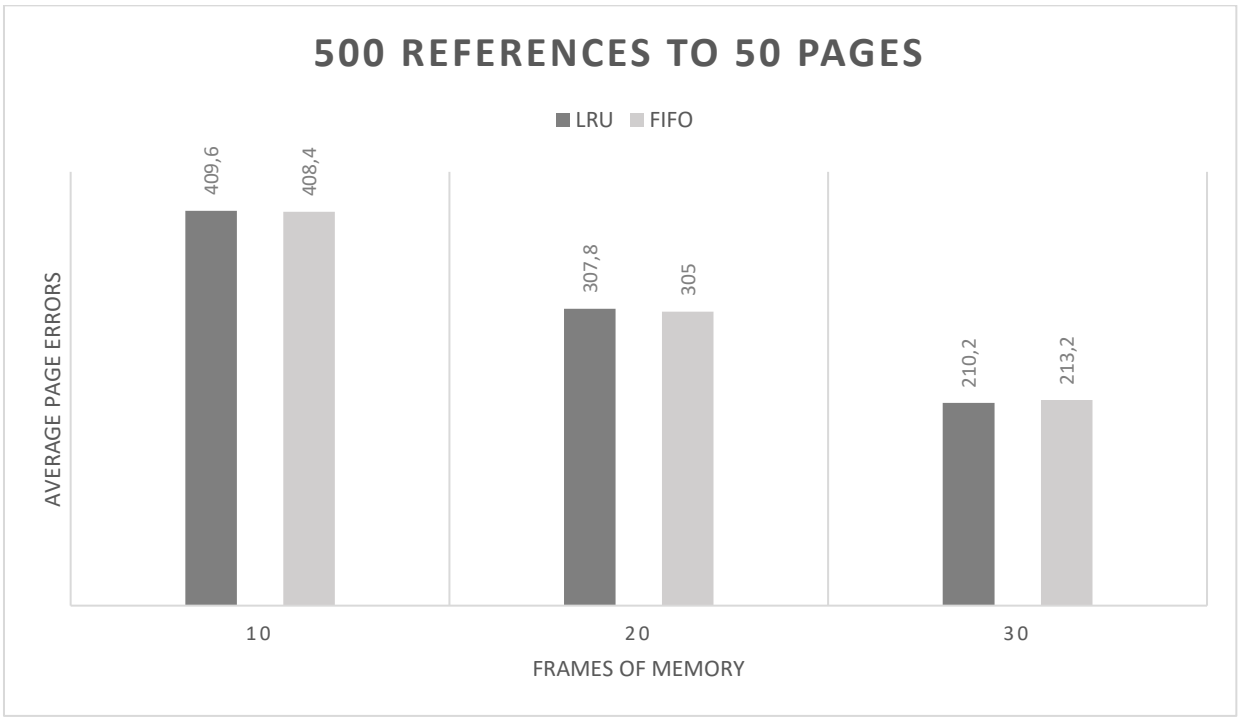


Wykres 2.1 15 odwołań do 10 różnych stron

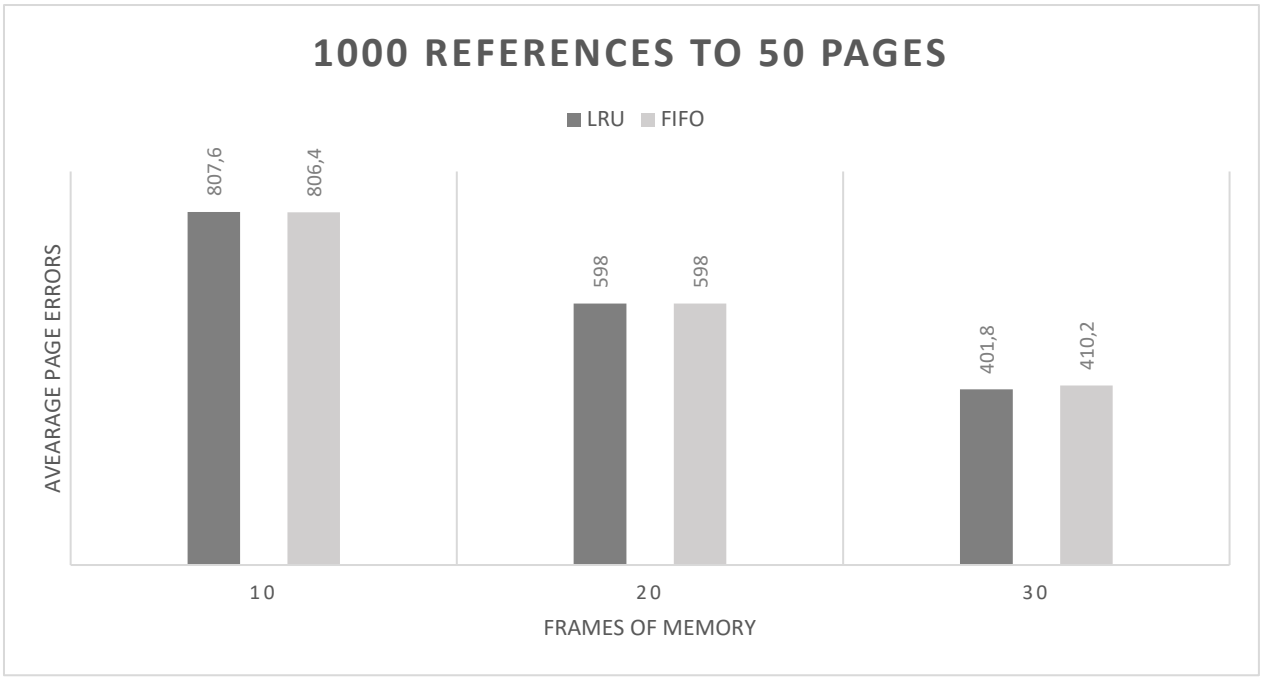
Wykres 2.2 50 odwołań do 10 różnych stron



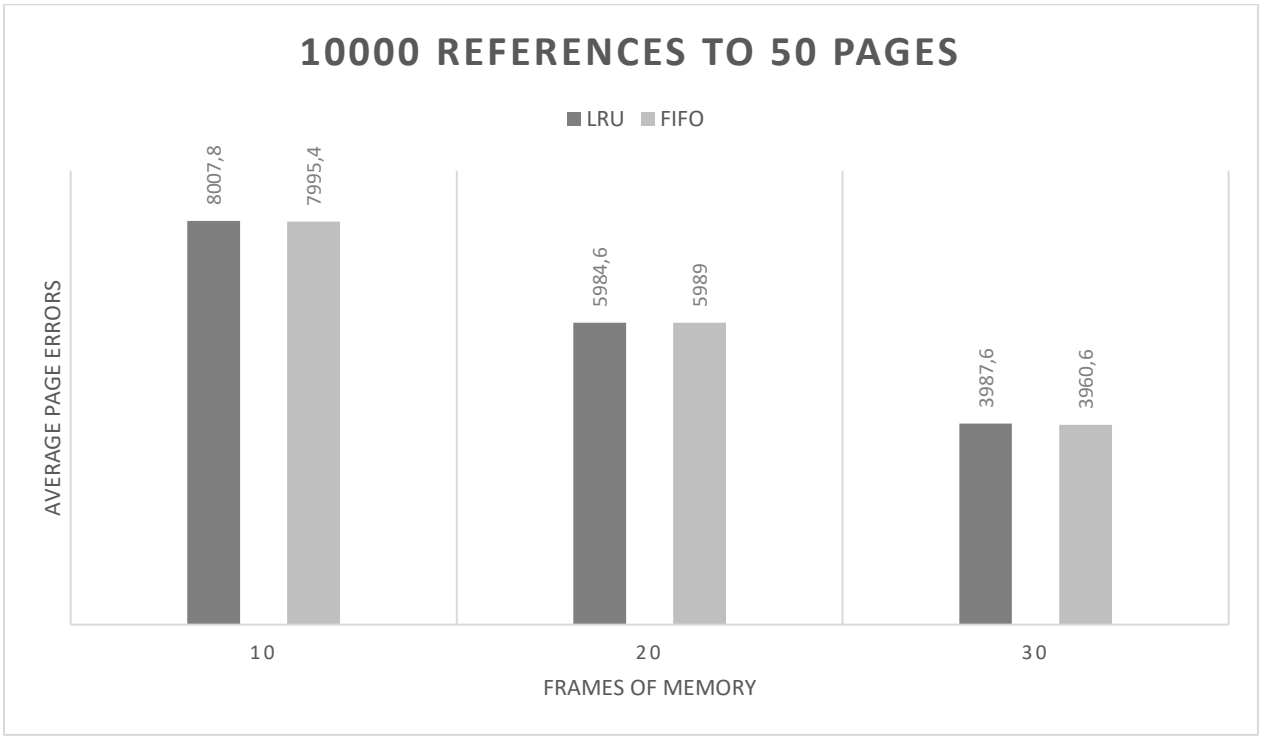
Wykres 2.3 100 odwołań do 10 różnych stron



Wykres 2.4 1000 odwołań do 100 róznych stron



Wykres 2.5 1000 odwołań do 50 róznych stron



Wykres 2.6 10000 odwołań do 50 róznych stron

2.5 Wnioski z eksperymentów

W miarę zwiększania liczby ramek obserwujemy zmniejszenie liczby błędów strony dla obu algorytmów. Jest to spodziewane zjawisko, gdyż większa ilość ramek oznacza większą przestrzeń do przechowywania aktywnych stron, co zmniejsza ryzyko błędów. Warto zauważyć, że w czwartym wywołaniu pojawiła się anomalia, gdzie zwiększenie liczby ramek spowodowało większą liczbę błędów strony. Wraz ze wzrostem liczby odwołań do pamięci obserwujemy znaczący wzrost liczby błędów strony zarówno dla algorytmu LRU, jak i FIFO. Wzrost ten wynika z intensywniejszej konkurencji o dostęp do ograniczonej liczby ramek pamięci w miarę zwiększania liczby odwołań. Dla mniejszej liczby ramek oba algorytmy wykazują porównywalne wyniki, ale LRU często osiąga minimalnie lepsze rezultaty. W miarę wzrostu liczby odwołań do pamięci obserwujemy znaczny wzrost błędów strony zarówno dla LRU, jak i FIFO. Wzrost ten wynika z zwiększonej konkurencji o dostęp do ramek pamięci, co staje się wyraźne w intensywniejszym użytkowaniu systemu. Algorytm LRU zdaje się być bardziej efektywny w redukcji błędów strony w porównaniu do FIFO, zwłaszcza w scenariuszach z mniejszą liczbą ramek. Dla bardzo dużych ilości odwołań do pamięci, oba algorytmy osiągają dość wysokie wartości błędów strony, co sugeruje trudność w skutecznym zarządzaniu pamięcią w przypadku intensywnego użycia. Wybór algorytmu zastępowania stron zależy od charakterystyki systemu i dostępności zasobów pamięciowych, nie ma jednoznacznego rozwiązania, oba algorytmy mają swoje zastosowania w różnych kontekstach.