Spis treści

Cel projektu

Głównym celem tego projektu była implementacja czterech wybranych algorytmów symulujących planowanie czasu procesora oraz algorytmów zastępowania stron. Dodatkowo konieczne było przetestowanie tych algorytmów dla różnych zestawów danych. W mojej pracy skupiłem się na algorytmach:

- FCFS – First Come Firs Serve
- SJF - Shortest Job Firs
- FIFO - First In First Out
- LRU - Least Recently Used

Opis kodu źródłowego

Kod projektu jest zapisany w języku Python z użyciem bibliotek Numpy, Matplotlib i Seaborn. Został on podzielony na osobne pliki – moduły: *klasa.py*, *generator_danych.py*, *FCFS.py*, *SJF.py*, *FIFO.py*, *LRU.py* oraz *main.py.* Każdy moduł zawiera funkcje niezbędne do swojego działania i może działać jako osobny program. Poniżej znajduje się opis niektórych funkcji.

Moduł klasa.py

```
class Proces:
    def __init__(self, nazwa, czas_przyjscia, czas_trwania):
        self.nazwa = nazwa
        self.czas_przyjscia = czas_przyjscia
        self.czas_trwania = czas_trwania
        self.czas_oczekiwania = 0
        self.start = czas_przyjscia

    def details(self):
        print("Process: " + self.nazwa + ", przyjscie: " +
str(self.czas_przyjscia) +
         ", czas trwania: " + str(self.czas_trwania) + str(self.czas_oczekiwania))
    def __repr__(self):
       return self.nazwa
```

Ten moduł został stworzony aby umożliwić przedstawienie pojedynczego procesy jako obiektu. Klasa Proces posada *atrybuty nazwa, czas_przyjscia, czas_trwania,* które są ustawiane przez generator lub ręcznie przez użytkownika. Każdy obiekt (proces), posiada również indywidualne atrybuty *czas_oczekiwania* oraz *start.* Są one używane przez algorytmy planowania czasu procesora.

Metoda *details()* może być opcjonalnie używana do wyświetlania informacji o danym obiekcie.

Metoda *__repr__* zwraca atrybut nazwy. Została ona zaimplementowana aby umożliwić czytelne wyświetlenie nazwy procesu na przykład poprzez funkcję *print()* lub debugger.

```
foo = Proces("p1", 0, 15)
print(foo)
#wyjscie z __repr__:   p1
#wyjscie bez __repr__: <__main__.Proces object at 0x000001DCBF3A9780>
```

Moduł generator_danych.py

Zadaniem tego modułu jest wygenerowanie czasów przyjścia i czasów trwania procesów. Generuje on również losowe sekwencje dostępu do pamięci.

```
def wygeneruj_czasy_trwania(Mu, Sigma, Size, id):
    czas_trwania = random.normal(Mu, Sigma, Size)
    plik = open("czasy_trwania" + id + ".txt", 'w')
    for i in czas_trwania:
        plik.write(str(abs(int(i)))+" ")
    plik.close()
```

Funkcja *wygeneruj_czasy_trwania(Mu, Sigma, Size)* generuje losowe czasy trwania procesów zgodnie z rozkładem normalnym, z podanymi parametrami średnią (Mu), odchyleniem standardowym (Sigma) i liczbą procesów (Size) i zapisuje je do pliku "czasy_trwania*[id]*.txt". Zapisywane dane wcześniej są zamieniane na liczby całkowite. Do tego celu należało zaimportować bibliotekę Numpy.

W analogiczny sposób generowane są czasy przyjścia, z tą różnicą, że dane losowane są z rozkładu jednostajnego. Sekwencje dostępu do pamięci generowane są przez funkcję *random.randint().*

Funkcja *pokaz_wykresy()* dodaje możliwość wyświetlenia wykresu dla czasów trwania i wykonywania procesów.

```
def DANE_RECZNE():
    procesy = []
    while True:
        wejscie = input("Podaj nazwe, czas przyjscia i trwania procesu:  ")
        if (wejscie == "0000"):
            break
        wejscie = wejscie.split(" ")
        procesy.append(klasa.Proces(wejscie[0], int(wejscie[1]),
int(wejscie[2])))
    return procesy
```

Funkcja *DANE_RECZNE()* pozwala na podanie danych procesów przez użytkownika. Użytkownik podaje odpowiednie dane oddzielając je spacją. Następnie ta informacja jest dzielona przez funkcję *split()* i odpowiednie wartości są dodawane do tablicy *procesy,* która przechowuje obiekty klasy Proces. Funkcja kończy swoją pracę gdy użytkownik wprowadzi wartość ‘0000’.

Funkcja *DANE_Z_PLIKU()* wczytuje dane procesów z pliku "*czasy_trwania.txt*" i "*czasy_przyjscia.txt*", odpowiednio je łączy oraz zwraca utworzoną z nich listę obiektów typu "Proces”.

Funkcja *SEKWENCJA_RECZNE()* pozwala na podanie sekwencji dostępu do pamięci przez użytkownika i zwraca listę z tymi sekwencjami

Funkcja *SEKWENCJA_Z_PLIKU()* wczytuje dane sekwencji z pliku "*sekwencja.txt*" i zwraca listę z tymi sekwencjami.

Moduł FCFS.py

```
def FCFS(procesy):
    procesy_sort = sorted(procesy, key=lambda x: x.czas_przyjscia)
    ilosc_procesow = len(procesy)

    czas = procesy_sort[0].czas_przyjscia
    suma_oczekiwan = 0

    for i in range (ilosc_procesow):
        if (procesy_sort[i].czas_przyjscia > czas):
            czas = procesy_sort[i].czas_przyjscia

        procesy_sort[i].start = czas
        czas = czas + procesy_sort[i].czas_trwania

        if (procesy_sort[i].start != 0):
            procesy_sort[i].czas_oczekiwania = procesy_sort[i].start -
procesy_sort[i].czas_przyjscia

        suma_oczekiwan = suma_oczekiwan + procesy_sort[i].czas_oczekiwania

    sredni_czas_oczekiwania = suma_oczekiwan / ilosc_procesow
    print("Średni czas oczekiwania dla algorytmu FCFS:    ",
sredni_czas_oczekiwania)
    return sredni_czas_oczekiwania
```

Moduł zawiera implementacje algorytmu First Come First Serve. Jego działanie polega na tym, że procesy są obsługiwane w kolejności ich przybycia. W pierwszym kroku program sortuje listę procesów właśnie po czasie przyjscia. Następnie dla każdego procesu ustawia czas startu (kiedy faktycznie zaczyna się wykonywać) jako sumę czasów trwania poprzednich procesów i oblicza czas oczekiwania jako różnicę między czasem startu a czasem przyjścia. Jeśli czas przyjścia procesu jest późniejszy niż obecnie przechowywany czas, aktualizujemy czas na czas przyjścia procesu. Na końcu obliczany jest średni czas oczekiwania dla wszystkich procesów.

Moduł SJF.py

```
def SJF(procesy):
    ilosc_procesow = len(procesy)
    gotowe = []
    aktualny_czas = 0
    suma_oczekiwan = 0
    procesy_sort = sorted(procesy, key=lambda x: x.czas_przyjscia)
    aktualny_czas = procesy_sort[0].czas_przyjscia

    i = 0
    procesy_sort = sorted(procesy, key=lambda x: x.czas_trwania)
    while procesy_sort:
        if (procesy_sort[i].czas_przyjscia <= aktualny_czas):
            procesy_sort[i].start = aktualny_czas
            aktualny_czas = aktualny_czas + procesy_sort[i].czas_trwania
            if (procesy_sort[i].start != 0):
                procesy_sort[i].czas_oczekiwania = procesy_sort[i].start -
procesy_sort[i].czas_przyjscia

            suma_oczekiwan += procesy_sort[i].czas_oczekiwania
            gotowe.append(procesy_sort.pop(i))
            i = 0
        else:
            i = i + 1
        if (i >= len(procesy_sort)):
            i = 0
            aktualny_czas = aktualny_czas + 1

    sredni_czas_oczekiwania = suma_oczekiwan / ilosc_procesow
    print("Średni czas oczekiwania dla algorytmu SJF:   ",
sredni_czas_oczekiwania)
    return sredni_czas_oczekiwania
```

Algorytm SJF (Shortest Job First) jest algorytmem planowania procesów, który polega na tym, że procesy są wykonywane w kolejności według ich czasów trwania. Procesy o mniejszym czasie trwania są wykonywane wcześniej niż te o dłuższym czasie trwania.

Kod implementuje algorytm SJF, który przyjmuje jako argument listę procesów i dla każdego z nich oblicza czas oczekiwania. Procesy są sortowane według czasów przyjścia (aby poznać czas przyjścia pierwszego procesu), a następnie według czasów trwania. W pętli while kod szuka procesu, który może zostać wykonany porównując aktualny czas i czas przyjścia procesu. Gdy taki znajdzie, zwiększa aktualny czas o jego czas trwania i przenosi go do listy gotowych procesów. W przypadku gdy kod nie znajdzie procesu możliwego do wykonania aktualny czas jest zwiększany o jeden. Kod powtarza swoje działanie do momentu gdy początkowa lista będzie pusta.

Czas oczekiwania procesów jest obliczany jako różnica między czasem rozpoczęcia a czasem przyjścia procesu.

Na końcu kod oblicza średni czas oczekiwania dla wszystkich procesów.

Moduł FIFO.py

```
def FIFO(sekwecja, wielosc_ramki):
    ramki = []
    bledy = 0

    for odwolanie in sekwecja:
        if odwolanie not in ramki: #czy dana wartosc znajduje sie w ramce
            bledy += 1
            if (len(ramki) == wielosc_ramki):
                ramki.pop(0) #usuniecie ostatniej wartosci
            ramki.append(odwolanie) #dodanie najnowszej
    print("Ilość błędów dla algorytmu FIFO:    ", bledy)
    return bledy
```

Moduł zawiera implementacje algorytmu FIFO zastępowania stron. Funkcja przyjmuje jako argumenty sekwencję odwołań do pamięci oraz rozmiar pamięci podręcznej. Algorytm przechodzi przez każde odwołanie w sekwencji i sprawdza, czy dane odwołanie znajduje się już w pamięci podręcznej. Jeśli tak, nic się nie dzieje. Jeśli nie, jest to błąd. W takim przypadku, jeśli pamięć podręczna jest już pełna, to najstarsze odwołanie jest usuwane, a nowe jest dodawane.

Na koniec algorytm zwraca liczbę błędów, które wystąpiły podczas przetwarzania sekwencji odwołań.

Moduł LRU.py

```
def LRU(sekwecja, wielosc_ramki):
    bledy = 0
    ramki = []
    for odwolanie in sekwecja:
        if odwolanie not in ramki:
            bledy += 1
            if (wielosc_ramki == len(ramki)):
                ramki.pop(0) #usuniecie ostatniej wartosci
            ramki.append(odwolanie) #dodanie najnowszej
        else:
            ramki.remove(odwolanie)
            ramki.append(odwolanie)
    print("Ilość błędów dla algorytmu LRU:    ", bledy)
    return bledy
```

Zasada działania algorytmu LRU jest podobna jak FIFO. Algorytm przechodzi przez każde odwołanie w sekwencji i sprawdza czy znajduje się już ono w pamięci. Gdy takiego odwołania nie ma, funkcja oznacza ten przypadek jako błąd i zapisuje odwołanie. Gdy odwołanie znajduje się już w pamięci, algorytm przenosi je na ostatnią pozycję. W ten sposób na pierwszej pozycji zawsze znajduje się najstarsze odwołanie, które następnie w razie potrzeby jest usuwane.

Moduł main.py

```
import generator_danych
import FCFS
import SJF
import FIFO
import LRU

suma_FCFS = 0
suma_SJF = 0
suma_FIFO = 0
suma_LRU = 0
ilosc_serii = 5
ramka = 4

for i in range(ilosc_serii):
    generator_danych.wygeneruj_czasy_trwania(60, 35, 200, str(i + 1))
    generator_danych.wygeneruj_czasy_przyjscia(0, 300, 200, str(i + 1))
    generator_danych.wygeneruj_sekwencje(15, 300, str(i + 1))

    procesy = generator_danych.DANE_Z_PLIKU(str(i + 1))
    sekwencja = generator_danych.SEKWENCJA_Z_PLIKU(str(i + 1))

    czas_fcfs = FCFS.FCFS(procesy)
    czas_sjf = SJF.SJF(procesy)
    print()
    bledy_fifo =FIFO.FIFO(sekwencja, ramka)
    bledy_lru = LRU.LRU(sekwencja, ramka)
    print()

    suma_FCFS += czas_fcfs
    suma_SJF += czas_sjf
    suma_FIFO += bledy_fifo
    suma_LRU += bledy_lru

avg_FCFS = round(suma_FCFS/ilosc_serii, 2)
avg_SJF = round(suma_SJF/ilosc_serii, 2)
avg_FIFO = suma_FIFO/ilosc_serii
avg_LRU = suma_LRU/ilosc_serii

print(avg_FCFS)
print(avg_SJF)
print()
print(avg_FIFO)
print(avg_LRU)
```

Ten moduł służy do przeprowadzenia testów symulacyjnych. Generuje on dane dla wybranych parametrów i wywołuje dla nich odpowiednie funkcje.

Przeprowadzone testy

Algorytmy planowania czasu procesora

W celu przetestowania wydajności algorytmów FCFS i SJF zostały przeprowadzone symulacje ich działania dla różnych parametrów takich jak: ilość procesów, czasy ich trwania oraz przyjścia o różnej rozpiętości. Ostatecznie wykonano badanie dla 5 zestawów danych każde po 5 serii.

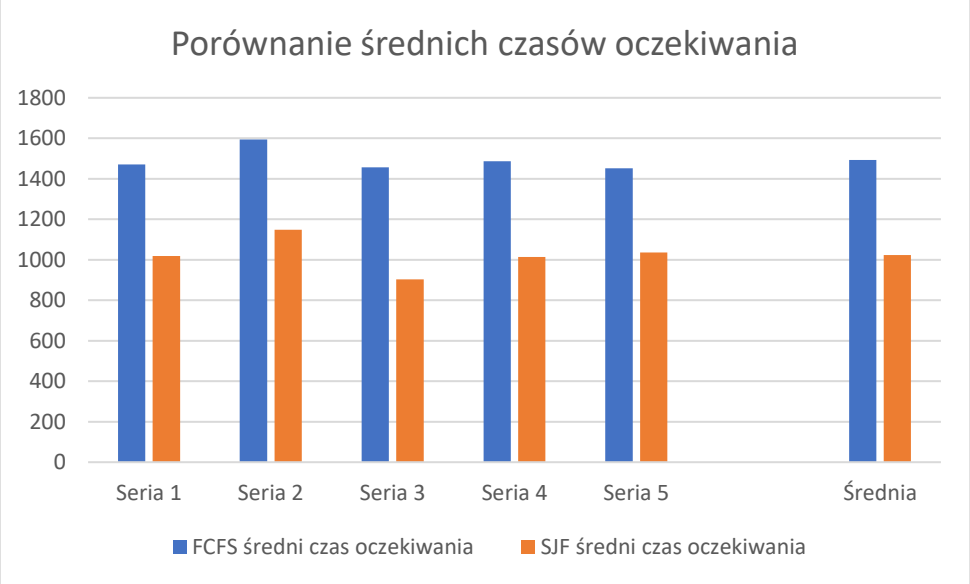
Wyniki cząstkowe

Pierwsza próba została przeprowadzona dla danych parametów:

- Ilość procesów: 100
- Czasy trwania:
  - Średnia: 30
  - odchylenie standardowe: 18
- Czasy przyjścia: zakres od 0 do 50

	FCFS średni czas oczekiwania	SJF średni czas oczekiwania
Seria 1	1471,23	1018,75
Seria 2	1594,19	1148,03
Seria 3	1457,29	902,29
Seria 4	1486,62	1013,63
Seria 5	1452,15	1035,7

Średnia	1492,30	1023,68

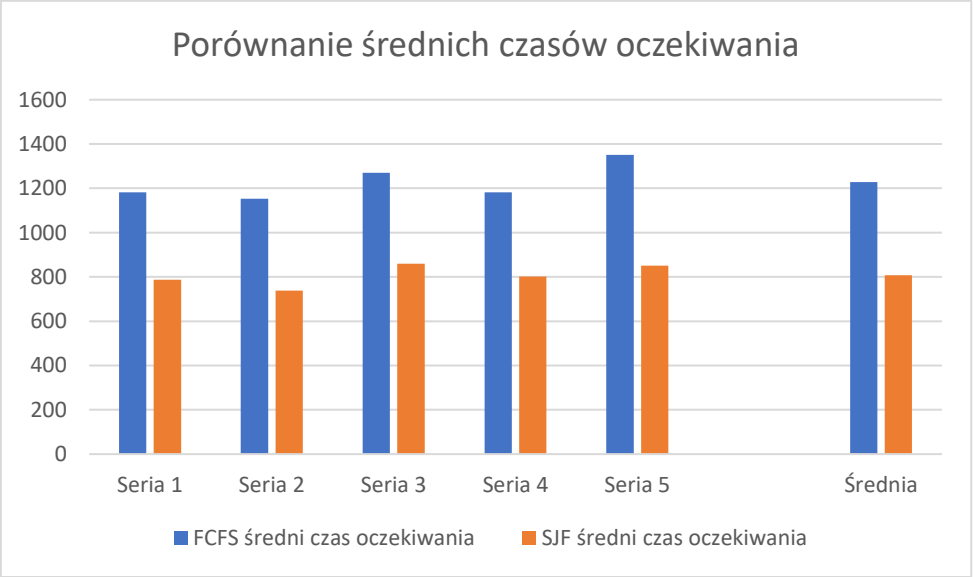


Druga próba została przeprowadzona dla danych parametów:

- Ilość procesów: 100
- Czasy trwania:
  - Średnia: 30
  - odchylenie standardowe: 18
- Czasy przyjścia: zakres od 0 do 500

	FCFS średni czas oczekiwania	SJF średni czas oczekiwania
Seria 1	1181,64	787,01
Seria 2	1152,97	737,98
Seria 3	1270,49	859,65
Seria 4	1182,1	802,16
Seria 5	1351,53	850,41

Średnia	1227,75	807,44

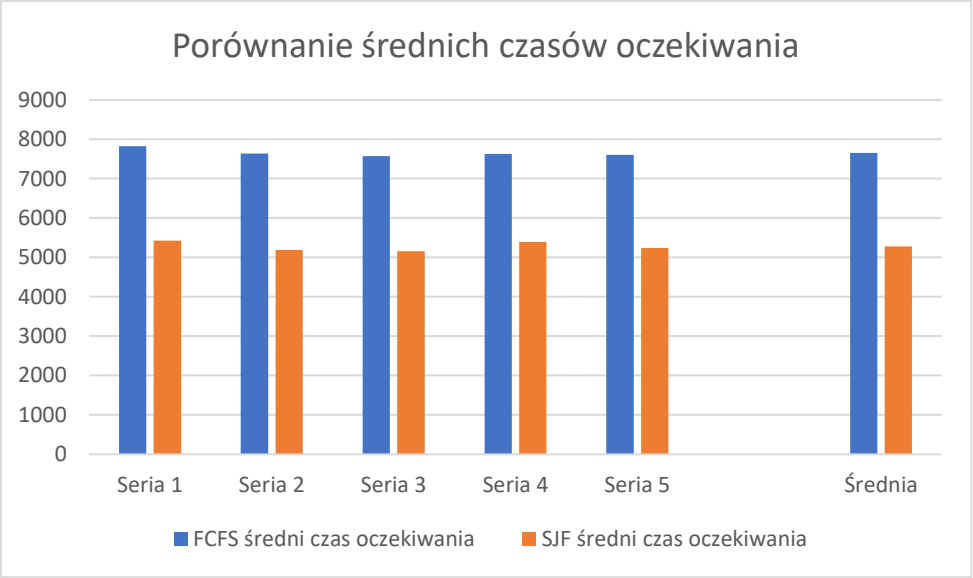


Trzecia próba została przeprowadzona dla danych parametrów:

- Ilość procesów: 500
- Czasy trwania:
  - Średnia: 30
  - odchylenie standardowe: 18
- Czasy przyjścia: zakres od 0 do 50

	FCFS średni czas oczekiwania	SJF średni czas oczekiwania
Seria 1	7828,99	5430,67
Seria 2	7637,51	5186,92
Seria 3	7577,05	5155,83
Seria 4	7629,48	5395,23
Seria 5	7602,24	5239,09

Średnia	7655,05	5281,55

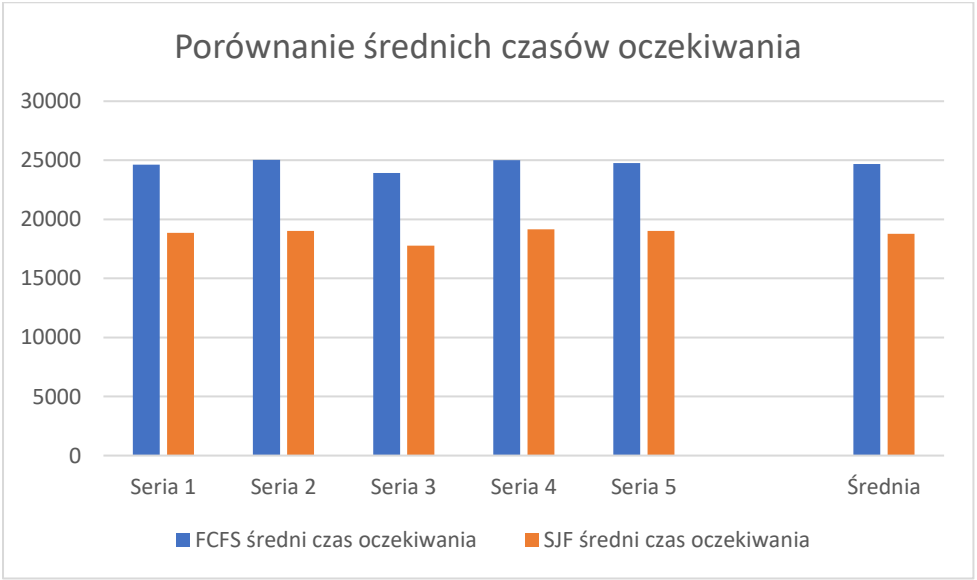


Czwarta próba została przeprowadzona dla danych parametów:

- Ilość procesów: 500
- Czasy trwania:
  - Średnia: 100
  - odchylenie standardowe: 44
- Czasy przyjścia: zakres od 0 do 500

	FCFS średni czas oczekiwania	SJF średni czas oczekiwania
Seria 1	24617,07	18844,92
Seria 2	25038,83	19023,16
Seria 3	23932,94	17784,21
Seria 4	25017,86	19159
Seria 5	24774,23	19012,39

Średnia	24676,19	18764,74

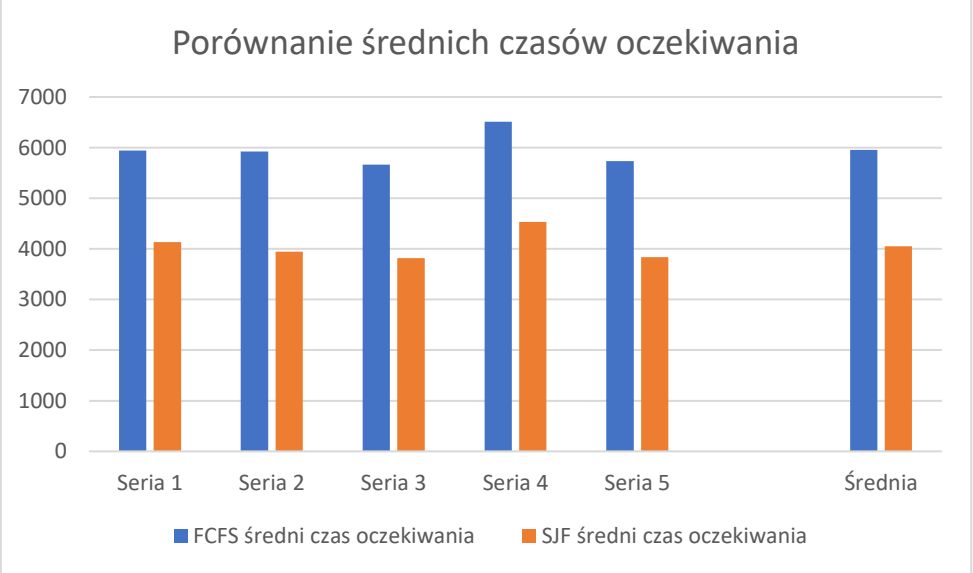


Piąta próba została przeprowadzona dla danych parametów:

- Ilość procesów: 200
- Czasy trwania:
  - Średnia: 60
  - odchylenie standardowe: 35
- Czasy przyjścia: zakres od 0 do 300

	FCFS średni czas oczekiwania	SJF średni czas oczekiwania
Seria 1	5942,01	4130,79
Seria 2	5925,64	3941,92
Seria 3	5665,09	3814,6
Seria 4	6508,62	4530
Seria 5	5730,36	3837,7

Średnia	5954,34	4051,00



Algorytmy zastępowania stron

Testowanie algorytmów FIFO i LRU polegało na wygenerowaniu sekwencji dla różnej ilość odwołań z różnego przedziału. Zmianie ulegała także pojemność ramki. Dla każdych 4 zestawów parametrów przeprowadzono 5 serii testowych.

Wyniki cząstkowe

Pierwsza próba została przeprowadzona dla podanych parametrów:

Pojemność ramki: 4
Ilość odwołań: 200
Ilość różnych odwołań: 10

Odwołanie:	0	1	2	3	4	5	6	7	8	9
Ilość wystąpień:	24	19	24	17	13	20	24	17	21	21

	Błędy FIFO	% Błędu	Błędy LRU	% Błędu
Seria 1	127	63,5%	130	65,0%
Seria 2	129	64,5%	132	66,0%
Seria 3	117	58,5%	116	58,0%
Seria 4	116	58,0%	111	55,5%
Seria 5	121	60,5%	119	59,5%

Średnia	122	61,0%	121,6	60,8%

Druga próba została przeprowadzona dla podanych parametrów:

Pojemność ramki: 8
Ilość odwołań: 200
Ilość różnych odwołań: 10

Odwołanie:	0	1	2	3	4	5	6	7	8	9
Ilość wystąpień:	18	22	22	16	27	23	22	17	17	16

	Błędy FIFO	% Błędu	Błędy LRU	% Błędu
Seria 1	54	27,0%	51	25,5%
Seria 2	53	26,5%	44	22,0%
Seria 3	48	24,0%	39	19,5%
Seria 4	46	23,0%	49	24,5%
Seria 5	46	23,0%	51	25,5%

Średnia	49,4	24,7%	46,8	23,4%

Trzecia próba została przeprowadzona dla podanych parametrów:

Pojemność ramki: 3
Ilość odwołań: 200
Ilość różnych odwołań: 10

Odwołanie:	0	1	2	3	4	5	6	7	8	9
Ilość wystąpień:	18	22	18	26	22	17	19	20	21	17

	Błędy FIFO	% Błędu	Błędy LRU	% Błędu
Seria 1	144	72,0%	143	71,5%
Seria 2	127	63,5%	125	62,5%
Seria 3	140	70,0%	141	70,5%
Seria 4	148	74,0%	146	73,0%
Seria 5	140	70,0%	139	69,5%

Średnia	139,8	69,9%	138,8	69,4%

Czwarta próba została przeprowadzona dla podanych parametrów:

Pojemność ramki: 4
Ilość odwołań: 300
Ilość różnych odwołań: 15

Odwołanie:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ilość wystąpień:	26	18	18	27	17	13	19	18	24	27	15	21	16	20	21

	Błędy FIFO	% Błędu	Błędy LRU	% Błędu
Seria 1	207	69,0%	209	69,7%
Seria 2	219	73,0%	216	72,0%
Seria 3	219	73,0%	218	72,7%
Seria 4	213	71,0%	219	73,0%
Seria 5	226	75,3%	228	76,0%

Średnia	216,8	72,3%	218	72,7%

Podsumowanie

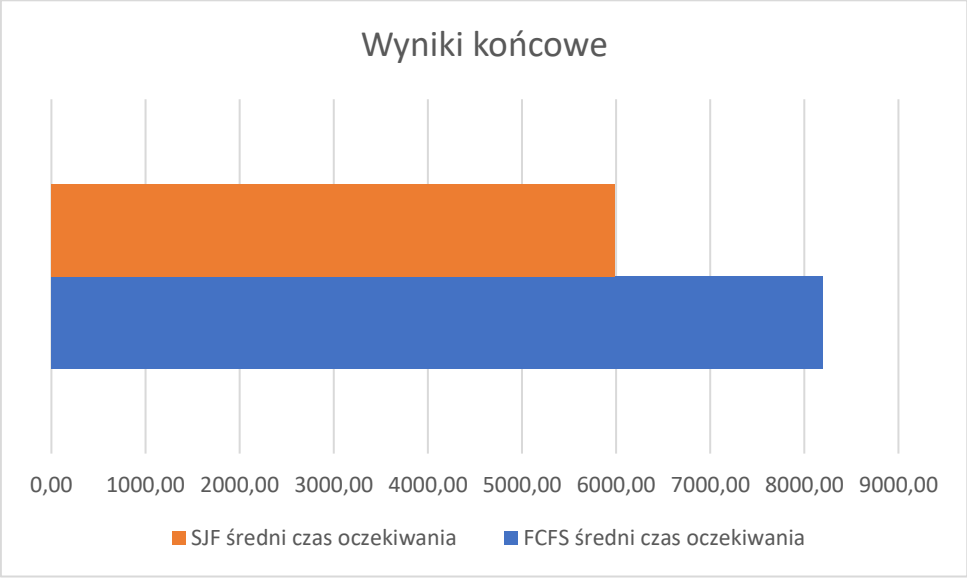
FCFS, SJF

Średni czas oczekiwania dla algorytmu FCFS zależy od kolejności, w jakiej procesy zostają przydzielone do procesora. Jeśli procesy o dłuższym czasie trwania są przydzielane jako pierwsze, to średni czas oczekiwania będzie dłuższy, ponieważ procesy o krótszym czasie trwania będą musiały czekać na swoją kolej. Nazywamy to efektem konwoju. Dla algorytmu SJF decydującym czynnikiem jest czas trwania procesu, im krótszy czas trwania tym mniejszy czas oczekiwania. Porównując wyniki z pierwszej i drugiej próby gdzie jedyną zmianą w parametrach wejściowych jest większy zakres losowości dla czasów przyjścia można zauważyć znaczącą poprawę wydajności obu algorytmów. Strategia FCFS zmniejszyła swój średni czas oczekiwania o 17,73% a SJF o 21,12%. Wynika to z faktu występowania przerw w dostarczaniu procesów, które nie wliczają się do czasów oczekiwania. Kolejnym istotnym czynnikiem jest sama ilość procesów. Można to zauważyć zestawiając wyniki pierwszej i trzeciej próby. W tym przypadku średni czas oczekiwania dla algorytmu FCFS odznaczył się wzrostem aż o 412,96% a SJF 415,93%. Sam czas trwania procesów jest równie znaczący. W czwartej próbie oscylował on wokół średniej wynoszącej 100 z odchyleniem standardowym równym 44. Zwiększona została również ilość procesów oraz ich czas przyjścia. Wynikiem tej kombinacji są średnie czasy trwania o największej wartości ze wszystkich prób. Jednak algorytm SJF okazał się być szybszy o 31,97%. Aby zminimalizować średni czas oczekiwania dla danych algorytmów należy dążyć do jak najmniejszej liczby procesów o jak najkrótszym czasie trwania.

Wyniki końcowe

Strategia „Shortest Job First” okazała się być bardziej wydajna we wszystkich pięciu próbach.

FCFS średni czas oczekiwania	8201,13
SJF średni czas oczekiwania	5985,68



FIFO, LRU

W przypadku algorytmu FIFO decydującym czynnikiem jest liczba ramek pamięci, które są dostępne dla algorytmu oraz kolejność odwołań do stron. Jeśli liczba ramek jest mniejsza niż liczba odwołań do stron, to algorytm będzie generował więcej błędów, ponieważ nie będzie miał miejsca, aby zapamiętać wszystkich odwołań. Algorytm FIFO polega na tym, że najstarsze odwołanie jest usuwane, gdy pojawia się nowe, więc odwołania, które są używane często, mogą być szybko usunięte z pamięci.

Algorytm LRU również bazuje na liczbie dostępnych ramek, ale w przeciwieństwie do FIFO, jest on oparty na zasadzie "najmniej ostatnio używanych" stron. Algorytm LRU zawsze usuwa stronę, która była używana najdawniej z pamięci, zastępując ją nową. Dzięki temu algorytm LRU jest bardziej skuteczny w radzeniu sobie z odwołaniami do stron, które są używane rzadko, ponieważ te strony są szybko usuwane z pamięci.

Niestety zastosowana metoda badawcza nie jest w pełnym stopniu poprawna. W przypadku losowego generowania stron, ilość błędów generowanych przez te algorytmy zależy od liczby ramek oraz od częstotliwości występowania poszczególnych stron w sekwencji odwołań. Można zauważyć to zestawiając wyniki drugiej i trzeciej próby. Dla drugiej próby zmniejszając liczbę ramek z 8 do 3 przy zachowaniu reszty parametrów średni procent błędu zwiększył się odpowiednio o 45,2 oraz 46 punktu procentowego. Ostatnia próba przedstawia sytuacje ekstremalną, gdzie stosunek pojemności ramki do liczby odwołań jest najmniejszy i występuje najwięcej różnych stron. Oba algorytmy uzyskały w tej próbie największy procent błędów, odpowiednio 72,3% oraz 72,7%.

Wyniki końcowe

Zarówno strategia FIFO jak i LRU uzyskały bardzo zbliżone wyniki końcowe z minimalną przewagą drugiego algorytmu.

FIFO średnia końcowa	132,00
LRU średnia końcowa	131,30

Średni % błędu FIFO	56,97%
Średni % błędu LRU	56,57%