Politechnika Wrocławska

Sprawozdanie z zadań Systemy Operacyjne

Piotr Modrakowski

Wydział Elektroniki, Systemy Operacyjne, Laboratoria, środa 17:05-18:45.Kod grupy:E04-70i

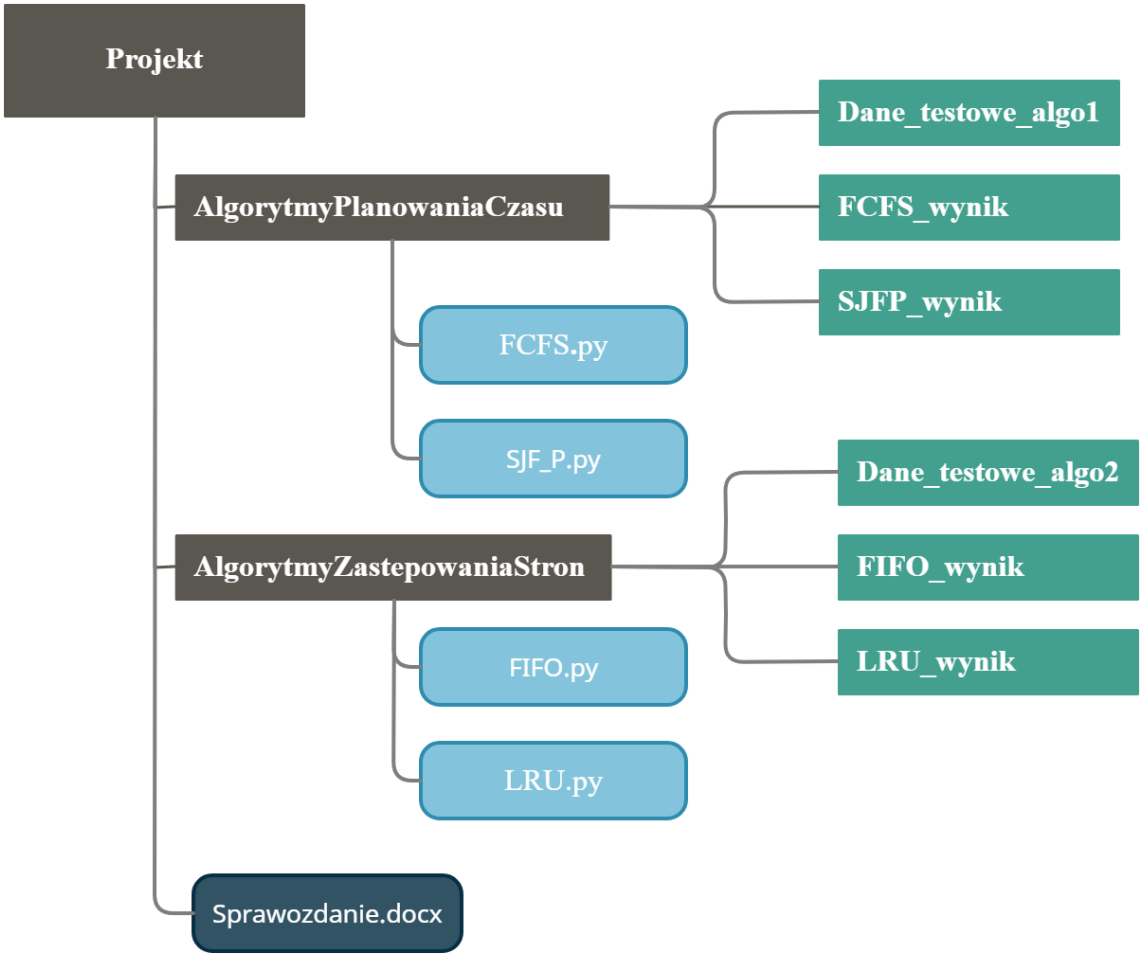
Semestr:2020/2021 Zimowy
Prowadzący: mgr inż. Jakub Klikowski

Wstęp do zadania.

Przedmowa

Przedmiotem zadań było przeprowadzenie symulacji działania algorytmów planowania czasu oraz algorytmów zastępowania stron. Dla procesów o określonych parametrach należało sprawdzić poprawność działania algorytmów.

Poniżej została umieszczona uproszczona struktura plików i folderów(rysunek 1), w celu łatwiejszego przemieszczania się po projekcie.



Rysunek 1-uproszczona struktura plików i folderów projektu

Algorytmy planowania czasu.

Dane testowe

Dane do testowania algorytmów zostały wygenerowane ze skryptu generator.py. Głównym założeniem było aby czas przybycia procesu był w zakresie od 0 do 19 jednostek

czasu, a czas wykonywania procesu trwał od 1 do 20 jednostek czasu. Zostały wygenerowane pliki z ilością procesów 10, 50, 250, 100, 1000 w 2 wariantach. Pierwszy wariant zakładał, że procesy przybywają w różnym czasie. W drugim wariancie zostało założone, że wszystkie procesy przybyły w jednym momencie, pliki te mają końcówkę arr0.txt.

Należy zaznaczyć, że czas trwania procesu w wariancie z różnym czasem przyjścia może się różnić od wariantu z jednakowymi czasami przyjścia, ponieważ losowania są nie zależne. Ponadto wszystkie procesy są rozróżnialne.

Kolejną uwagą do danych testowych jest to, że procesy w wariancie o różnych czasach przyjścia mogą się powtarzać, to znaczy że możemy mieć na przykład 5 procesów o czasie przyjścia 2 jednostki czasu i 10 procesów o czasie przyjścia 7. Takie „grupowanie" się procesów w zależności od czasu przyjścia jest normalnym zjawiskiem i nie powoduje żadnych błędów, które mogłyby wykluczyć jakiś zbiór procesów. Najczęściej występuje przy dużej ilości procesów np. 1000. Jednak nie jest powiedziane, że nie wystąpi to dla tablic z mniejszą ilością procesów np.10 lub 5.

Generator danych testowych

Skrypt *proces_generator.py* przyjmuje argument z rozszerzeniem txt, który jest nazwą pliku z surowymi danymi do algorytmu.
Składania skryptu:

python generator.py <nazwa_pliku_z_danymi_testowymi.txt>

Po uruchomieniu skryptu możemy wprowadzić liczbę procesów, którą chcemy wygenerować oraz czy czasy przyjścia mają być różne od siebie czy takie same. Dla jednakowych czasów przyjścia czas przyjścia dla każdego procesu wynosi 0. Można ten czas zmienić w kodzie generator.py, ale jest to równoznaczne z tym, że czas przetwarzania wszystkich procesów będzie dłuższy o czas przybycia. Zakres czasu wykonywania procesów wynosi od 1 do 20. Skrypt do generowania czasów przybycia i trwania korzysta z biblioteki random. Aby stworzyć odpowiednią strukturę do przechowywania danych czasowych została zastosowana biblioteka numpy. Struktura procesu jest tablicą dwu elementową, gdzie pierwszy element w tablicy przechowuje czas przybycia a w drugim mamy czas trwania procesu. Wygląda następująco:

tablica_procesów=[[czas_przybycia, czas_trwania], …]

Tworzenie listy procesów, etapy:

1. Na początku jest uruchamiany skrypt i użytkownik podaje potrzebne dane, takie jak na rysunku 2:

```
Podaj ilosc procesow do wygenerowania: 10
Czy chcesz aby procesy przyszly w tym samym momencie(TAK=1,NIE=0): 0
```

Rysunek 2

2. Następnie są generowane procesy za pomocą wywołania funkcji *proc_list(zero_arrival)*. Zmienna *zero_arrival* jest wprowadzana w 2 pytaniu do użytkownika. Procesy po wygenerowaniu są sortowane w zależności od czasu przybycia. Od najkrótszego czasu przybycia do najdłuższego. Wszystkie wygenerowane argumenty są przypisywane do pustej tablicy o rozmiarach *(size,2)*, gdzie *size* to ilość procesów do wygenerowania podana w 1 zapytaniu skryptu.

3. Przed ostatnim etapem jest zapisanie tablicy z procesami *proces_que* do pliku z argumentu 1 skryptu w formacie numpy. Nazwa pliku musi być podana z rozszerzeniem .txt, aby skrypty poprawnie odczytały tablice z wygenerowanymi procesami. Zapisanie odbywa się poprzez wywołanie funkcji *save_data(proces_que).*

```
def save_data(process_que):
    f=open(sys.argv[1],"w")
    for i in process_que:
        np.savetxt(f,i)
    f.close()
```

Rysunek 3-funkcja zapisująca tablice do pliku

4. Ostatnim etapem jest wyświetlenie tablicy wygenerowanych procesów w terminalu. Plik z wygenerowanymi danymi możemy znaleźć w tym samym folderze co skrypt.

Procedura testowania

Plik z algorytmem przyjmuje dwa argumenty w postaci pliku z danymi w formacie txt oraz nazwę plików wynikowych. Pierwszym argumentem skryptu jest lista danych testowych. Kolejny argument jest to nazwa pliku do którego są zapisywane dane z eksperymentu, który powstanie w wyniku działania skryptu. Składnia skryptu do konsoli wygląda tak:

python <testowany_skrypt.py > < nazwa_pliku_z_danymi_testowymi.txt > <nazwa_pliku_z_wynikami_eksperymentu >

Skrypt pobiera dane z pliku z danymi, następnie przetwarza w określony w sposób. Wyniki są zapisywane w nowo utworzonych plikach .txt i .xls o nazwę podaną w drugim argumencie skryptu. Aby skrypt działał poprawnie należy mieć zainstalowaną python'owską bibliotekę xlwt oraz numpy. Ponadto plik ze skryptem musi być umieszczony razem z plikami z danymi testowymi w jednym folderze. W projekcie można zauważyć inną strukturę, ponieważ uległy ręcznemu posortowaniu w celu łatwiejszego poruszania się po projekcie.

Pliki wynikowe

W pliku z rozszerzeniem .txt mamy zapisany całkowity przebieg eksperymentu. W pliku z rozszerzeniem .xls mamy procesy i wraz z danymi o czasach: przybycia, trwania,

oczekiwania, całkowity czas trwania procesu. Ponadto obok tabeli z poszczególnymi procesami zostały umieszczone takie dane jak średni czas czekania procesów, średni czas trwania procesów oraz całkowity czas przetwarzania wszystkich procesów. Następnie dla średnich czasów czekania i trwania procesów zostały wyliczone odchylenia standardowe, w obydwu przypadkach zastosowano zaokrąglenie do 3 miejsc po przecinku.

FCFS – First Come First Served

FCFS jest jednym z algorytmów planowania czasu procesora. Polega on na tym, że proces który pierwszy przybędzie zostanie on w pierwszej kolejności obsłużone przez procesor. Pozostałe, które przybyły później aż pierwszy skończy wykonywać swoją pracę. O kolejności obsłużenia decyduje czas przybycia.

Przy tworzeniu poniższych tabel z wynikami eksperymentów wzięto pod uwagę średnie czasy oczekiwania oraz trwania procesów wraz wyliczonymi odchyleniami standardowymi w zależności od ilości procesów.

1. Wyniki eksperymentu dla procesów z różnymi czasami przybycia.

Ilość procesów	Średni czas oczekiwania procesów	Odchylenie standardowe
10	36,5	23,936
50	233,94	139,605
100	496,34	298,478
250	1326,296	756,817
1000	5213,176	2980,685

Ilość procesów	Średni czas trwania procesów	Odchylenie standardowe
10	46,3	25,118
50	244,2	140,357
100	506,6	298,798
250	1336,964	756,761
1000	5223,482	2980,448

2. Wyniki eksperymentu dla procesów przybyłych w jednym momencie.

Ilość procesów	Średni czas trwania procesów	Odchylenie standardowe
10	45,6	31,472
50	247,74	131,761
100	522,76	315,503
250	1348,228	791,839
1000	5339,526	3031,752

Ilość procesów	Średni czas trwania procesów	Odchylenie standardowe
10	55,5	32,078
50	257,04	130,86
100	533,49	315,964
250	1358,916	791,708
1000	5350,162	3031,693

SJF wywłaszczeniowy

SJF wywłaszczeniowy jest kolejnym algorytmem planowania czasu procesora. Polega on na tym, że w pierwszej kolejności są obsługiwane zadania o najkrótszym czasie trwania, mają one absolutny priorytet. Są uprzywilejowane do tego stopnia, że przerywa pracę procesu o dłuższym czasie trwania i automatycznie są mu przyznawane zasoby procesora.

Przy tworzeniu poniższych tabel z wynikami eksperymentów wzięto pod uwagę średnie czasy oczekiwania oraz trwania procesów wraz wyliczonymi odchyleniami standardowymi w zależności od ilości procesów.

1. Wyniki eksperymentu dla procesów z różnymi czasami przybycia.

Ilość procesów	Średni czas oczekiwania procesów	Odchylenie standardowe
10	29,1	27,493
50	167,86	151,907
100	350,09	308,176
250	910,424	794,346
1000	3479,646	3049,784

Ilość procesów	Średni czas trwania procesów	Odchylenie standardowe
10	40,2	31,615
50	178,22	157,11
100	360,49	313,427
250	921,12	799,896
1000	3489,965	3055,419

2. Wyniki eksperymentu dla procesów przybyłych w jednym momencie.

Ilość procesów	Średni czas oczekiwania procesów	Odchylenie standardowe
10	34	28,449
50	146,78	139,188
100	359,67	324,51

250	922,54	799,981
1000	3633,999	3172,684

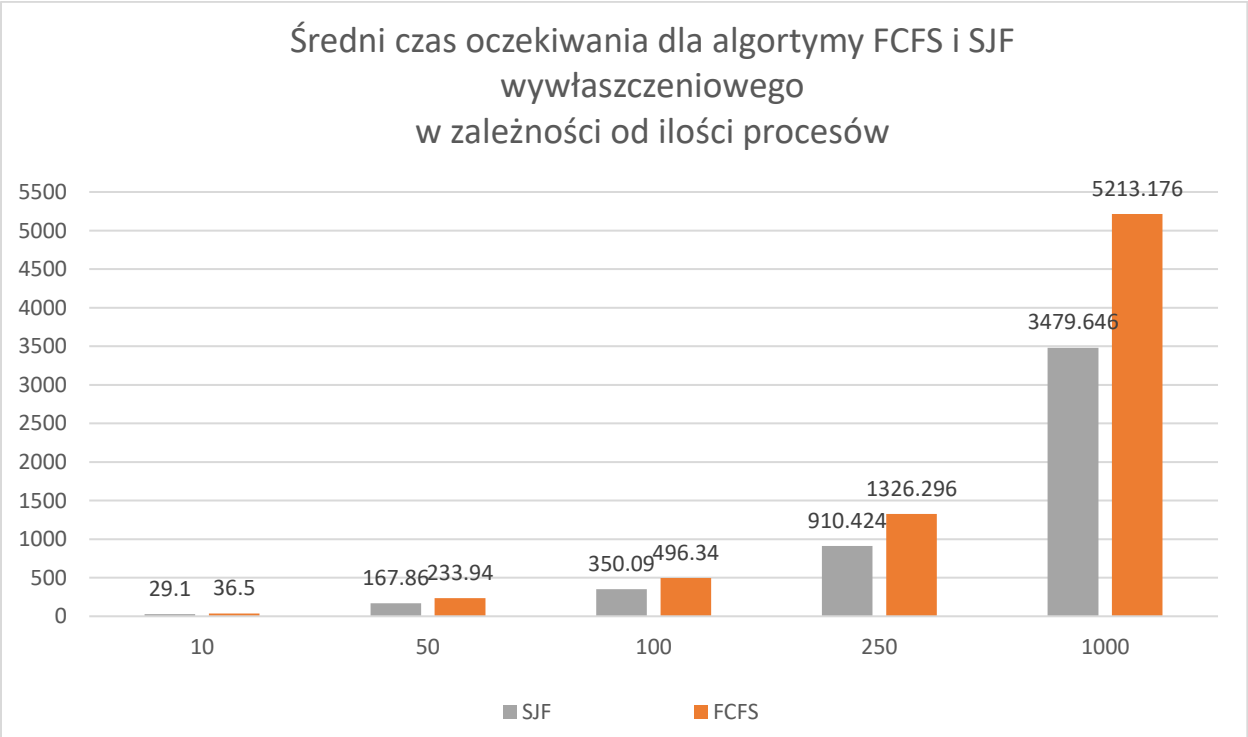
Ilość procesów	Średni czas trwania procesów	Odchylenie standardowe
10	44,5	33,444
50	156,4	145,323
100	370,5	330,404
250	933,292	805,576
1000	3644,655	3178,356

Podsumowanie i wnioski.

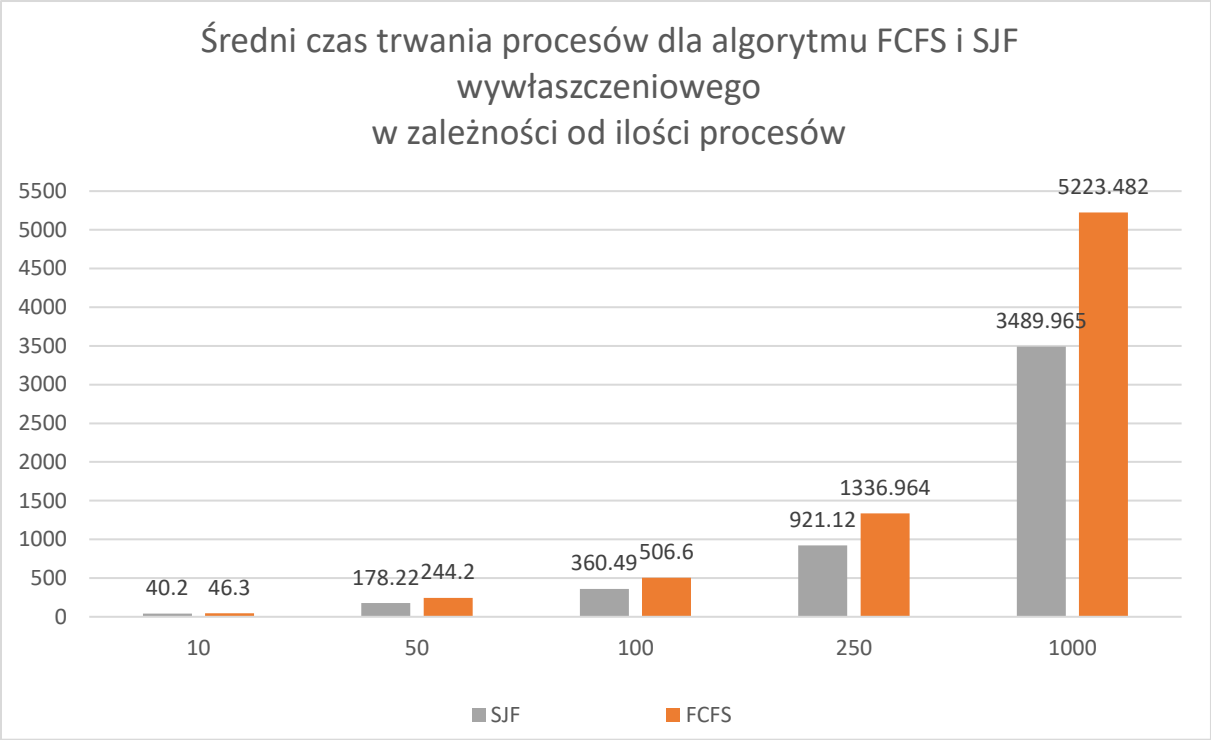
Na podstawie powyższych tabel zostały sporządzone wykresy słupkowe w celu lepszej wizualizacji danych i wyciągnięcia wniosków. Wykresy zostały sporządzone dla 2 różnych wariantów wymienionych w podrozdziale **„Dane testowe”**: procesy o różnych czasach przyjścia i o jednakowych. Należy wspomnieć, że im mniejszy całkowity czas oczekiwania i trwania tym algorytm jest wydajniejszy

Wykresy dla różnych czasów przyjścia.

- Średni czas czekania procesów.

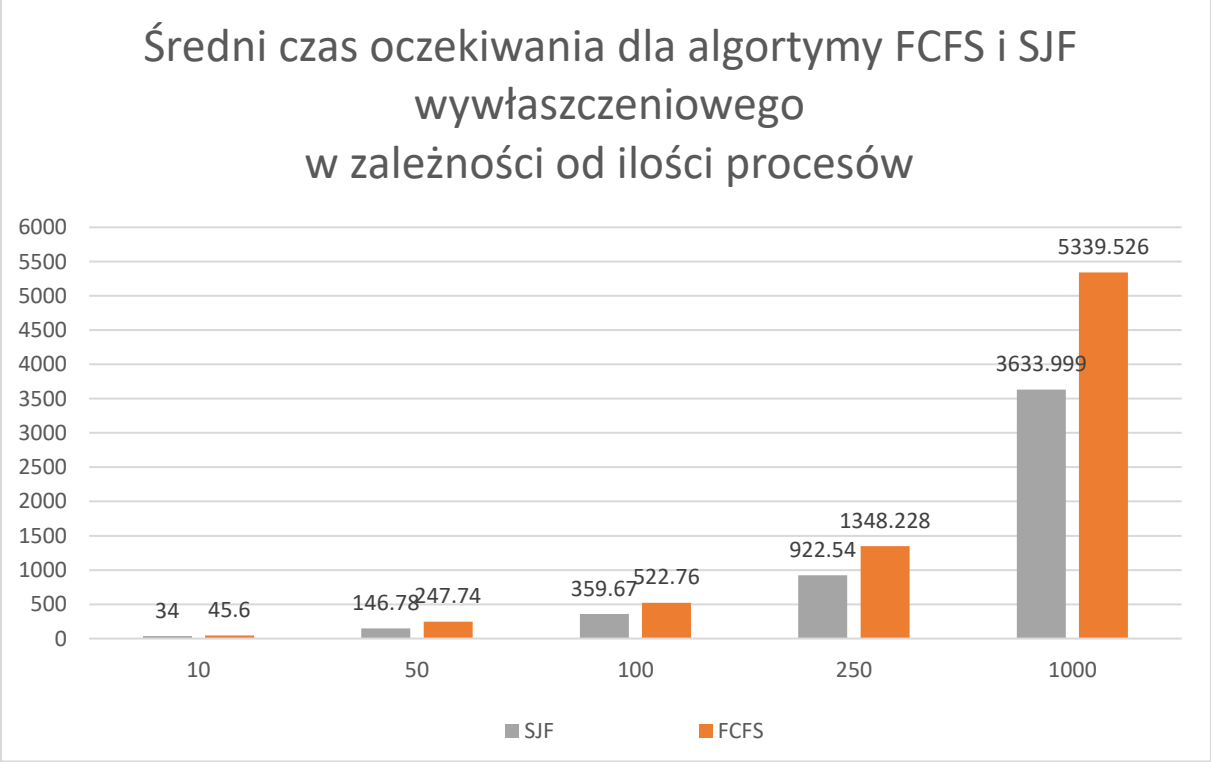


- Średni czas trwania procesów.

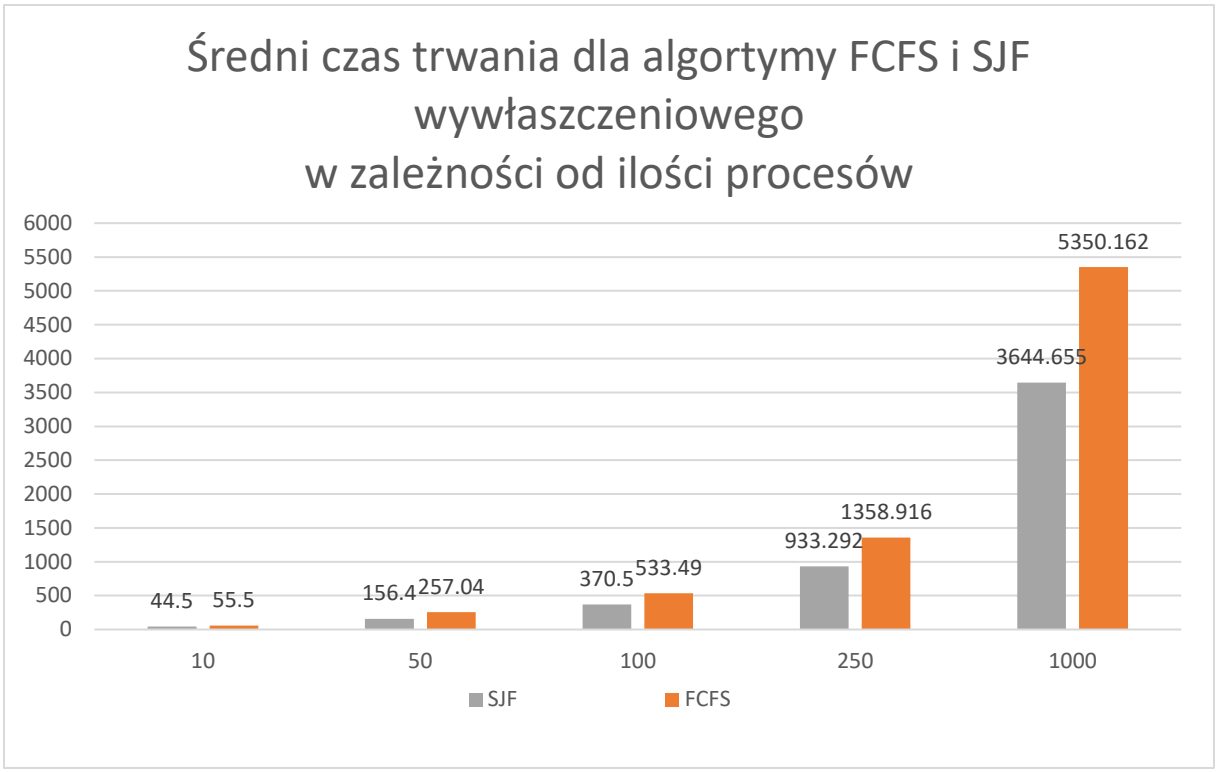


Wykresy dla jednakowych czasów przyjścia.

- Średni czas czekania procesów.



- Średni czas trwania procesów.



Analizując średnie i biorąc pod uwagę odchylenia standardowe średnich dla czasu całkowitego trwania i czekania, można zauważyć, że wraz ze wzrostem ilości procesów jest znaczący wzrost odchylenia standardowego. Wynika to z większej mnogości i różnorodności procesów, jednak największy wpływ mają wady związane ze stosowaniem badanych algorytmów.

W przypadku stosowania algorytmu FCFS mamy sytuację kolejkowania procesów(efekt konwoju). Oznacza to, że proces o długim czasie trwania, który przybył pierwszy i zostały mu przydzielone zasoby procesora, blokuje inne. Procesy, które przybyły bezpośrednio po nim muszą poczekać aż proces pierwszy wykona swoją pracę, co się wilcza do ich czasu oczekiwania, a finalnie ma wpływ na średni czas oczekiwania jak i trwania. Przy większej ilości nastąpi zgrupowanie procesów o tym samym czasie przyjścia. Nasuwa się pytanie, który proces zostanie obsłużony jako pierwszy z grupy o jednakowym czasie przyjścia? Zostanie obsłużony ostatnio dodany proces z zgrupowanej listy(została zastosowana metoda LIFO). Wynika to z tego, że lista wygenerowanych procesów ze skryptu gnerator.py ma ustawioną kolejność, która jest losowa. Z tego też wynika, że kolejkowanie może być cykliczne, to oznacza że dla każdej grupy procesów o tym samym czasie przyjścia może nastąpić problem konwojowania. Planowanie tą metodą jest nieoptymalne.

Przypadku algorytmu SJF wywłaszczeniowego następuje problem związany z zwiększonym czasem oczekiwania procesu o długim czasie trwania na rzecz procesów o krótkim czasie trwania. Jeśli w czasie trwania procesu, nadejdzie proces krótszy to nastąpi jego wywłaszczenie. Dla dużej ilości procesów może się wydarzyć, że proces o długim czasie trwania może być przerwany znaczącą liczbę razy przez procesy o krótszym czasie trwania, co zwiększy jego czas oczekiwania, co ma później wpływ na średni czas oczekiwania oraz trwania procesów. Jednak długi czas trwania i oczekiwania procesu jest równoważony przez krótsze czasy oczekiwania procesów wywłaszczonych, które miały krótsze czasy trwania i

przerywały proces długi. Układ ten jest korzystniejszy ponieważ algorytm wypada korzystniej w większości wypadków(ilością procesów wprowadzonych do symulacji) względem FCFS. Niemniej jednak skrajne czasy mają zły wpływ na wiarygodność średniej trwania i czekania procesów, ponieważ wynik te są bardziej oddalone od średniej jeśli spojrzymy na odchylenie standardowe dla SJF. Z drugiej strony nie są to na tyle duże różnice, aby całkowicie zaprzeczyć lepsze osiągi dla algorytmu SJF.

Algorytmy zastępowania stron.

Dane testowe

Dane do testowania algorytmów zostały wygenerowane ze skryptu page_generator.py. Dane zostały wygenerowane w dwóch wariantach o numeracji stron o zakresie numerowania stron od 0 do 10 oraz od 0 do 15. Ilość stron dla obu wariantów była następująca: 10, 20, 30, 40.

Generator danych testowych

Skrypt page_generator.py przyjmuje argument z rozszerzeniem txt, który jest nazwą pliku z surowymi danymi do algorytmu.
Składania skryptu:

python page_generator.py <nazwa_pliku_z_danymi_testowymi.txt>

Po uruchomieniu skryptu możemy wprowadzić liczbę stron, którą chcemy wygenerować oraz w jakim zakresie mają być numerowane. Dla jednakowych czasów przyjścia czas przyjścia dla każdego procesu wynosi 0. Można ten czas zmienić w kodzie generator.py, ale jest to równoznaczne z tym, że czas przetwarzania wszystkich procesów będzie dłuższy o czas przybycia. Zakres czasu wykonywania procesów wynosi od 1 do 20. Skrypt do generowania numerów stron korzysta z biblioteki random. Aby stworzyć odpowiednią strukturę do przechowywania danych czasowych została zastosowana biblioteka numpy. Struktura procesu wygląda tak:

Tworzenie listy procesów, etapy:

1. Na początku jest uruchamiany skrypt i użytkownik podaje potrzebne dane, takie jak na rysunku 4:

```
Podaj ilość stron do wygenerowania: 10
Podaj zakres losowania numeracji stron: 10
```

Rysunek 4

2. Następnie są generowane strony za pomocą wywołania funkcji *page_list()*. Wszystkie wygenerowane argumenty są przypisywane do pustej tablicy *pages*, która jest zmienną globalną.

3. Przed ostatnim etapem jest zapisanie tablicy z procesami *pages* do pliku z argumentu 1 skryptu, dane z tablicy są zapisywane w formacie zgodnym z biblioteką numpy. Nazwa pliku musi być podana z rozszerzeniem .txt, aby skrypty poprawnie odczytały tablice z wygenerowanymi procesami. Zapisanie odbywa się poprzez wywołanie funkcji *save_data(pages).*

4. Ostatnim etapem jest wyświetlenie tablicy wygenerowanych stron w terminalu. Plik z wygenerowanymi danymi możemy znaleźć w tym samym folderze co skrypt.

Procedura testowania

Plik z algorytmem przyjmuje 3 argumenty. Pierwszym argumentem skryptu jest lista danych testowych. Kolejnym argumentem jest ilość ramek, jest to liczba całkowita. Ostatni argument to nazwa pliku wynikowego. Składnia skryptu do konsoli wygląda tak:

python <testowany_skrypt.py > < nazwa_pliku_z_danymi_testowymi.txt > <rozmiar_ramki> <nazwa_pliku_z_wynikami_eksperymentu >

Skrypt pobiera dane z pliku z danymi, następnie przetwarza w określony w sposób. Wyniki oraz przebieg jest zapisywany w nowo utworzonym pliku .xls o nazwę podaną w trzecim argumencie skryptu. Aby skrypt działał poprawnie należy mieć zainstalowaną python'owską bibliotekę xlwt oraz numpy. Ponadto plik ze skryptem musi być umieszczony razem z plikami z danymi testowymi w jednym folderze. W projekcie można zauważyć inną strukturę, ponieważ po wykonaniu operacji uległy ręcznemu posortowaniu w celu łatwiejszego poruszania się po projekcie.

Pliki wynikowe

W wynikowym arkuszu kalkulacyjnym mamy umieszczone wynik naszego eksperymentu. W pierwszym wierszu mamy załadowane w odpowiedniej kolejności wchodzące strony. W kolumnie A w wierszach od F1 do Fn mamy ramki. W ostatnim wierszu tabeli mamy informacje o tym czy doszło do wymiany strony. Wymiana strony jest oznaczona jest jako duża litera F, duża litera H oznacza, że dana strona została już załadowana i nie doszło do błędu wymiany strony. Dane pod tabelą sumują błędy gdzie doszło do wymiany strony oraz te gdzie do wymiany nie dochodziło, bo strona już została załadowana do pamięci. Pod poniższymi danymi został obliczony współczynnik błędu wymiany strony(page fault error rate) podany w procentach, jego wzór wygląda następująco:

$$współczynnik\ błędu\ wymiany\ stron = \frac{wszystkie\ błędy\ wymiany\ stron}{ilość\ wszystkich\ stron} \times 100$$

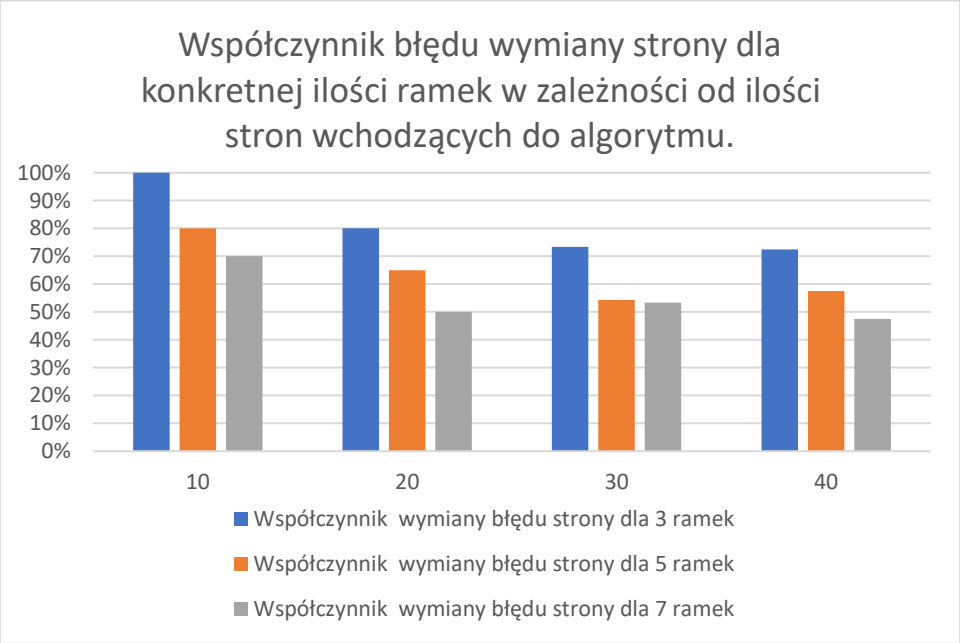
FIFO – first in first out

Algorytm FIFO polega na tym, że strona pozostająca najdłużej w pamięci jest pierwsza do wymiany w przypadku dojścia do błędu wymiany strony. Strony w pamięci fizycznej tworzą kolejkę FIFO, która działa tak, że strony ostatnio dodane są sprowadzane na koniec kolejki, a ‘strony-ofiary’ są na jej początku i są brane do wymiany w przypadku błędu wymiany.

Przy tworzeniu poniższych tabel z wynikami eksperymentów wzięto pod uwagę współczynnik błędu wymiany strony dla zadanej liczby ramek w zależności od ilości stron wchodzących do algorytmu. Na podstawie tabel został sporządzony wykres słupkowy, który został umieszczony pod odpowiadającą mu tabelą.

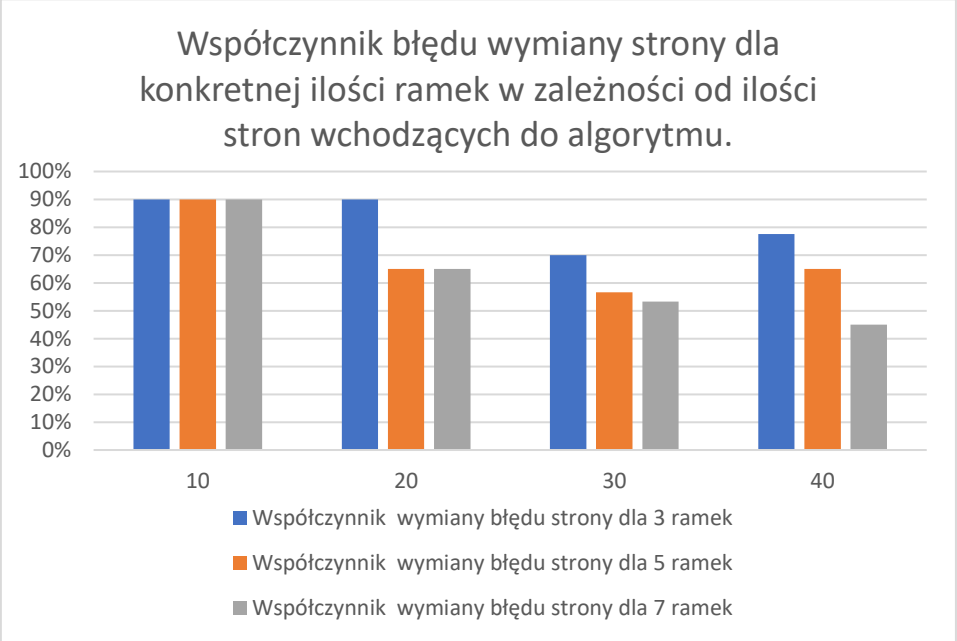
1. Wyniki eksperymentu dla zakresu numeracji od 0 do 10.

Ilość stron	Współczynnik wymiany błędu strony dla 3 ramek	Współczynnik wymiany błędu strony dla 5 ramek	Współczynnik wymiany błędu strony dla 7 ramek
10	100%	80%	70%
20	80%	65%	50%
30	73,33%	54,33%	53,33%
40	72,5%	57,49%	47,5%



2. Wyniki eksperymentu dla zakresu numeracji stron od 0 do 15.

Ilość stron	Współczynnik wymiany błędu strony dla 3 ramek	Współczynnik wymiany błędu strony dla 5 ramek	Współczynnik wymiany błędu strony dla 7 ramek
10	90%	90%	90%
20	90%	65%	65%
30	70%	56,66%	53,33%
40	77,5%	65%	45,0%



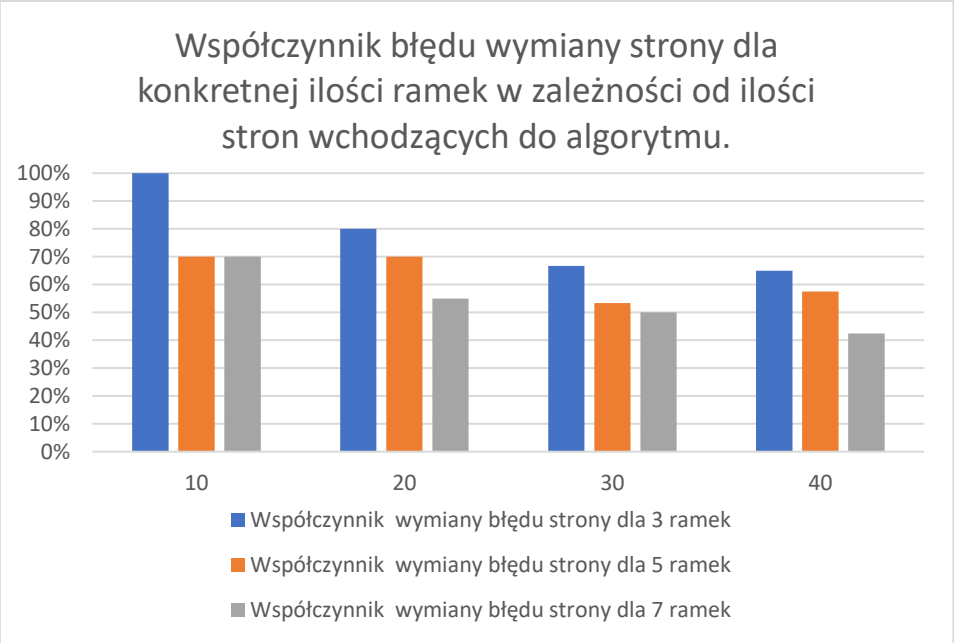
LRU – last recently used

W algorytmie LRU w przypadku wystąpienia braku strony, zostaje wymieniona strona, do której się nie odwoływano przez najdłuższy okres.

Przy tworzeniu poniższych tabel z wynikami eksperymentów wzięto pod uwagę współczynnik błędu wymiany strony dla zadanej liczby ramek w zależności od ilości stron wchodzących do algorytmu. Na podstawie tabel został sporządzony wykres słupkowy, który został umieszczony pod odpowiadającą mu tabelą.

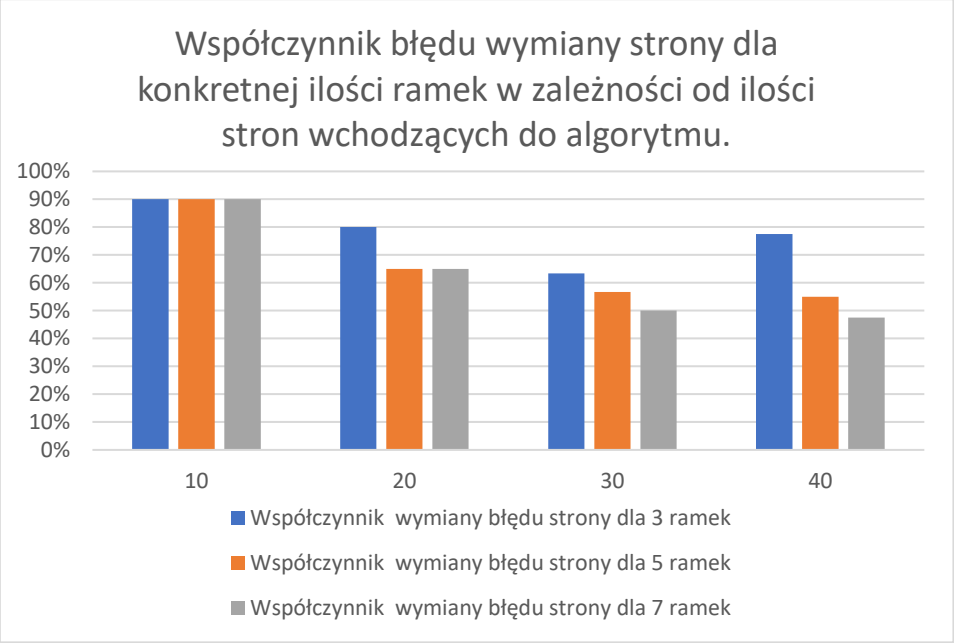
1. Wyniki eksperymentu dla zakresu numeracji od 0 do 10.

Ilość stron	Współczynnik wymiany błędu strony dla 3 ramek	Współczynnik wymiany błędu strony dla 5 ramek	Współczynnik wymiany błędu strony dla 7 ramek
10	100%	80%	70%
20	80%	65%	50%
30	73,33%	54,33%	53,33%
40	72,5%	57,49%	47,5%



2. Wyniki eksperymentu dla zakresu numeracji stron od 0 do 15.

Ilość stron	Współczynnik wymiany błędu strony dla 3 ramek	Współczynnik wymiany błędu strony dla 5 ramek	Współczynnik wymiany błędu strony dla 7 ramek
10	90%	90%	90%
20	90%	65%	65%
30	70%	56,66%	53,33%
40	77,5%	65%	45,0%

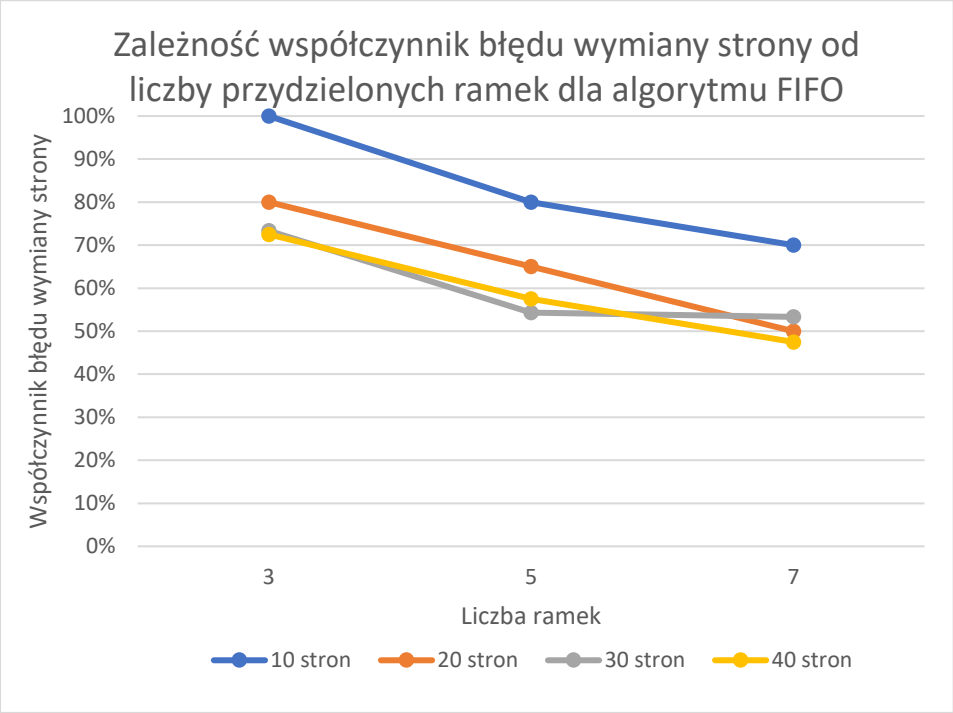


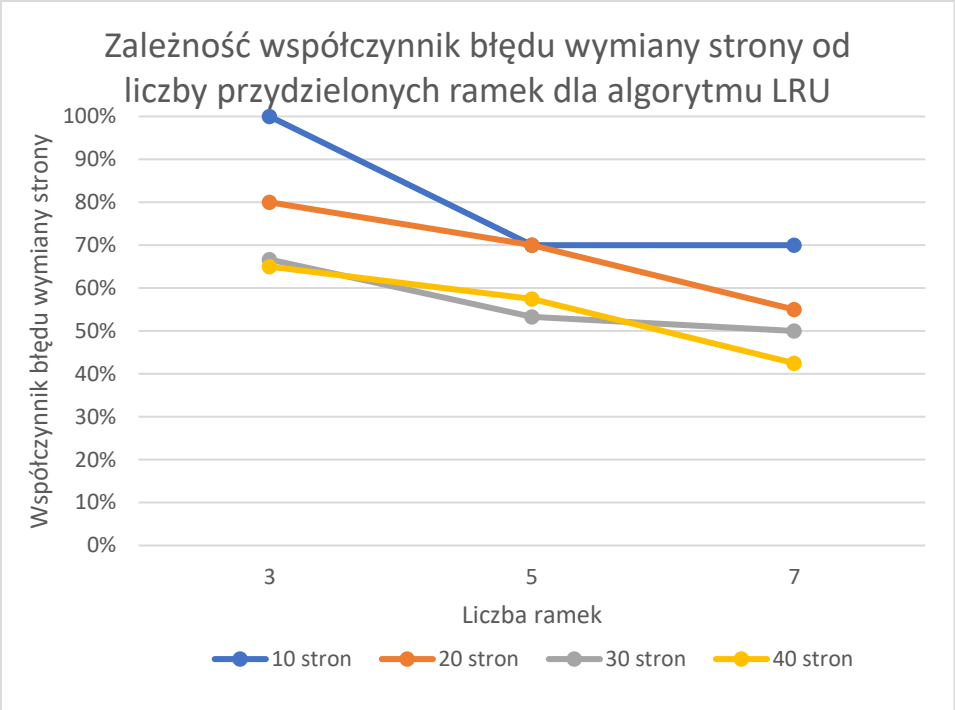
Podsumowanie i wnioski.

Na podstawie powyższych tabel o został sporządzony wykres liniowy ze znacznikami współczynnika błędu wymiany strony w zależności od liczby ramek, które ukazuje anomalie Belady'ego, jeśli owa faktycznie wystąpiła.

Wyżej wymieniona anomalia ukazuje zjawisko, że w niektórych algorytmach współczynnik błędu wymiany strony może ulec wzrostowi wraz ze wzrostem liczby wolnych ramek.

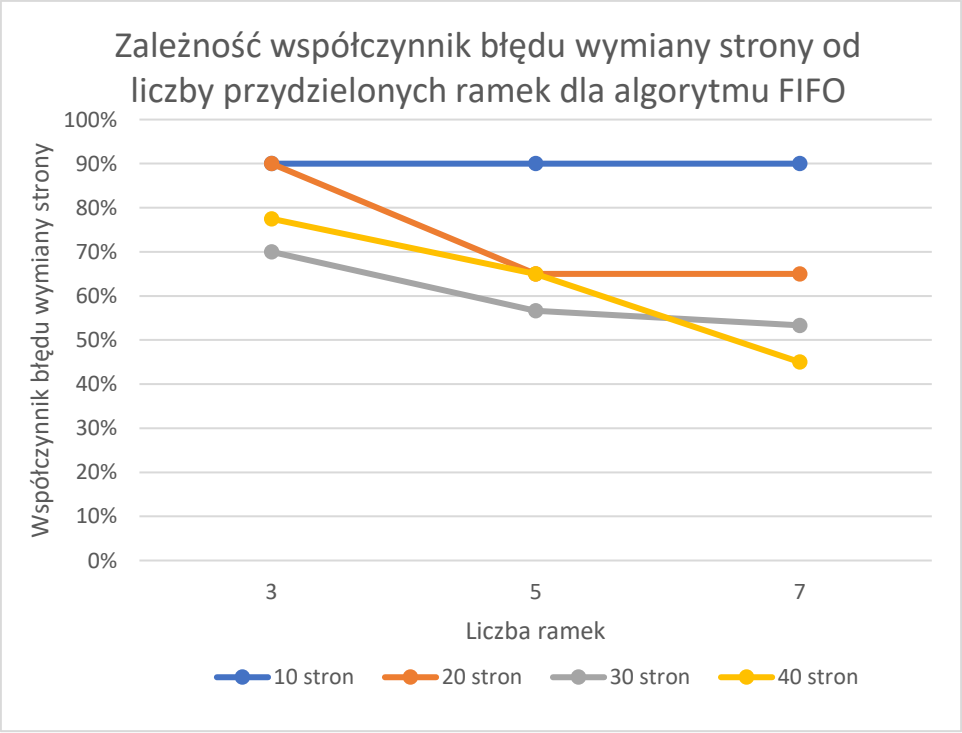
1. Badanie czy wystąpiła anomalia Belady'ego dla algorytmy FIFO i LRU z zakresu numeracji stron od 0 do 10.

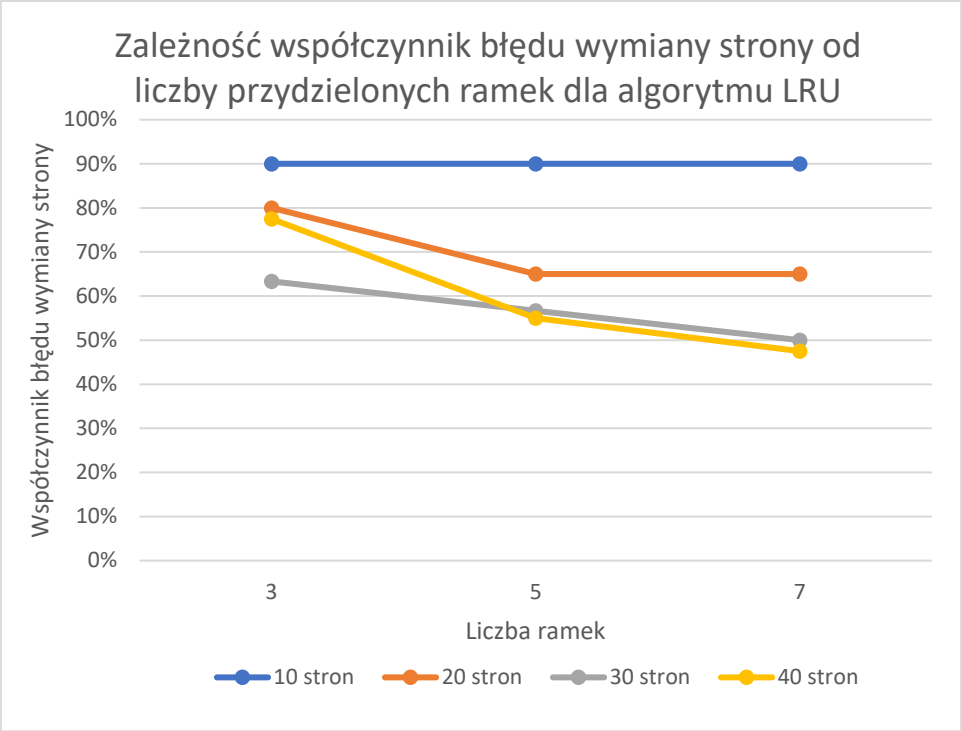




W żadnym w wyżej badanych przypadków w tym podpunkcie nie doszło do anomalii Balady'ego.

2. Badanie czy wystąpiła anomalia Belady'ego dla algorytmy FIFO i LRU z zakresu numeracji stron od 0 do 15.





W żadnym w wyżej badanych przypadków w tym podpunkcie nie doszło do anomalii Balady'ego.

W przypadku algorytmu FIFO w dwóch wariantach numeracji stron nie doszło do anomalii w obu podpunktach. Tak samo było przypadku algorytmu LRU nie doszło do anomalii i nie powinno, bo oznaczałoby to brak poprawności działania algorytmu LRU oraz złą jego implementacje. Algorytm LRU jest odporny na anomalie Balady'ego.

Jak już wcześniej zostało wspomniane, do anomalii może dojść, ale nie musi. Zostały wylosowane takie dane, które nie wywołały tego szczególnego przypadku w algorytmie FIFO.

Podsumowując działanie obu algorytmów na podstawie tabel oraz wykresów dla dwóch osobnych przypadków numeracji stron, można dojść do pewnych wniosków co do korzyści i wad ich stosowania.

Pierwszą uwagą jest to, że dla wszystkich przypadków ilości stron współczynniki błędu wymiany strony w niektórych przypadkach są takie same lub różnią się od siebie w nieznacznym stopniu. Raz lepiej wypada FIFO, a innym razem lepszy współczynnik błędu wymiany strony ma LRU. Należy tu zaznaczyć, że im mniejszy współczynnik tym uzyskujemy lepszy rezultat. Również należy wspomnieć, aby nie zestawiać ze sobą współczynników dla różnych ilości stron, ponieważ daje to mylny obraz na skuteczność algorytmu. W większości przypadków współczynnik dla LRU jest niższy niż dla FIFO, jednak nie jest to jednoznaczne z tym, że algorytm LRU jest zdecydowanie lepszy. Należy pamiętać, że algorytm LRU jest trudniejszy w implementacji i w niektórych przypadkach uzyskamy małą skuteczność dużym kosztem. Kosztem tym jest implementacja dodatkowej struktury przechowującej ostatnio wchodzące numery stron do wymiany. Warto się wtedy zastanowić nad algorytmem sortującym strony FIFO.

Bibliografia

- https://docs.python.org/3/library/random.html
- https://numpy.org/
- https://www.w3schools.com/python/numpy_intro.asp
- https://pypi.org/project/xlwt/
- https://edu.pjwstk.edu.pl/wyklady/sop/scb/wyklad8/wyklad.html
- https://www-users.mat.umk.pl//~zssz/PSO2001/Wyk9/tsld022.html
- „Podstawy systemów operacyjnych”-Avi Silberschatz