Sprawozdanie

Przeprowadzenie symulacji działania planowania czasu procesora oraz zastępowania stron

Oświadczam, że osobiście wykonałem to sprawozdanie.
Patryk Wesołowski, 283879,Cyberbezpieczeństwo 2024/2025

Spis treści

Wstęp

1. Opis modułów

Program zostały stworzony w oparciu o 4 moduły:

a. main.py – moduł główny przeprowadzający symulację, testy
b. cpuschedule.py – zawiera zaimplementowane algorytmy kolejkowania procesów oraz generator procesów
c. pagereplacement.py – zawiera zaimplementowane algorytmy zastępowania stron oraz generator stron
d. graphs.py – odpowiada za rysowanie wykresów dla każdego testu

Program przygotowuje podstawowe wykresy na podstawie wyników, ale przedstawione wykresy zostały utworzone w MS Excel.

2. Zastosowane algorytmy planowania czasu procesora

    a. Shortest Job First(SJF)
    b. Round Robin(RR)

3. Zastosowane algorytmy zastępowania stron

    a. First In First Out(FIFO)
    b. Least Frequently Used(LFU)
    c. Least Recently Used(LRU)

Symulacja planowania czasu procesora

1. Opis działania algorytmów SJF i RR

Shortest Job First – algorytm kolejkuje procesy według czasu wykonania. Czas procesora zostanie przydzielony temu procesowi, który w danym momencie czasu będzie mógł wykonać się jak najszybciej.

W porównaniu do innych metod możemy zaobserwować skrócony średni czas oczekiwania.

Największą z wad filozofii tego algorytmu jest to, że należy z góry znać czasy wykonania dla każdego procesu oraz to, że procesy o wysokich wymaganiach odnośnie czasu mogą doznać *zagłodzenia,* szczególnie, gdy będzie przychodzić wiele procesów o krótkich czasach wykonania. Dodatkowo może zajść odwrotna sytuacja i procesy o znacznie krótszych czasach wykonania będą czekały dłużej(efekt konwoju).

Round Robin – algorytm opiera się na założonym z góry maksymalnym czasie przydziału procesora dla każdego procesu. Jeśli proces nie zdążył wykonać się przed wywłaszczeniem trafia on na koniec kolejki.

Zalety opierają się na tym, że każdy proces dostaje równy czas dostępu do procesora, co implikuje brak występowania efektu zagłodzenia oraz konwoju procesów. Co więcej, jest łatwy jest zaimplementowania.

Wadą tego algorytmu jest to, że nie jest zbyt wydajny. Przy złym dobranym kwancie czasu, wykonanie niektórych procesów zajmie dużo więcej czasu. Ponad to każde przełączenie procesu(kontekstu) zajmuje dodatkową moc obliczeniową oraz powoduje straty w czasie.

2. Opis procedury testowania

Procesy są generowane za pomocą funkcji *generate_processes()* (rysunek 1.) z modułu *cpuschedule.py*. Funkcja zwraca listę procesów, posortowaną według czasu nadejścia.

```
def generate_processes(num,arrival_range=(0,1),burst_range=(0,1),seed=None): # Generator procesów
    random.seed(seed)
    return sorted([Process(pid=i, arrival_time=random.randint(arrival_range[0], arrival_range
    [1]), burst_time=random.randint(burst_range[0], burst_range[1])) for i in range(num)],
    key=lambda x: x.arrival_time)
```

Rysunek 1

W główny module możemy ustawić opcje odnośnie generowania procesów dla każdego testu (Rysunek 2.). Możemy również ustalić ziarno dla powtarzalności wyników. Ustalamy również nazwę folderu, w którym zapisywane będą katalogi z wynikami.

```
#Ustawienia symulacji kolejkowania procesów
settings_cpu = {
    "Number Of Processes": [25,75,125,250],
    "Arrival Range": [(0,100) for i in range(4)],          # Ustawienia generowania procesów
    "Burst Range": [(5,200) for i in range(4)],
    "Quantum": [5 for i in range(4)]
}
test_number = min(len(v) for v in settings_cpu.values())  # Liczba testów
test_dir="CPUTests" # Folder do zapisu wyników poszczególnych testów
```

Rysunek 2

Dla każdego testu tworzymy osobną strukturę plików (Rysunek 3.)

```
for t in range(test_number):
    if os.path.exists(f"test{t}"):
        shutil.rmtree(f"test{t}")
    os.makedirs(f"test{t}", )
    os.chdir(f"test{t}")
```

Rysunek 3

Każdy test zostaje uruchomiony funkcją *run_cpu_tests(),* która przyjmuje ustawienia odnośnie poszczególnego testu. Zostają w niej wygenerowane procesy, które następnie zostają przekazane do funkcji

simulation_cpu_schedulding(), która uruchamia symulację dla zaimplementowanych algorytmów (Rysunek 4.).

```
def run_cpu_tests(settings_cpu, t,seed):  #Funkcja obsługująca dany test dla algorytmów planowania

    cpu_algorithms = {
        "Shortest Job First": sjf,
        "Round Robin": partial(round_robin,quantum=settings_cpu["Quantum"][t])
    }

    processes = generate_processes(     # Generowanie procesów uzależnionych od ustawień dla konkretnego testu
        settings_cpu["Number Of Processes"][t],
        settings_cpu["Arrival Range"][t],
        settings_cpu["Burst Range"][t],
        seed
        )
    results_cpu = simulation_cpu_schedulding(cpu_algorithms,processes) # Zwrot wyników symulacji do dalszej analizy

    return results_cpu
```

Rysunek 4

Funkcja *simulation_cpu_schedulding()* uruchamia bezpośrednio każdy z zaimplementowanych algorytmów. Jako wynik pobiera kolejność czasu wykonywania procesów oraz średnie czasy realizacji i oczekiwania, następnie informacje te zostają zapisane do konkretnego pliku. Sama funkcja zwraca tylko i wyłącznie średnie czasy, które zostaną użyte do analiz.

```python
def simulation_cpu_schedulding(algorithms,processes): # Przeprowa
    average_times = {} # Przechowanie wyników do analizy dla każd
    for name, alg in algorithms.items():
        schedulded_list, times = alg(processes)  # Uruchomienie s
        average_times[name] = times #
        with open(f"{name}.txt",'w') as file: # Zapisujemy do pli
            file.write("PID;START;END\n")
            for process in schedulded_list:
                out=f"{process[0]};{process[1]};{process[2]}\n"
                procesora
                file.write(out)
    return average_times
```

Rysunek 5

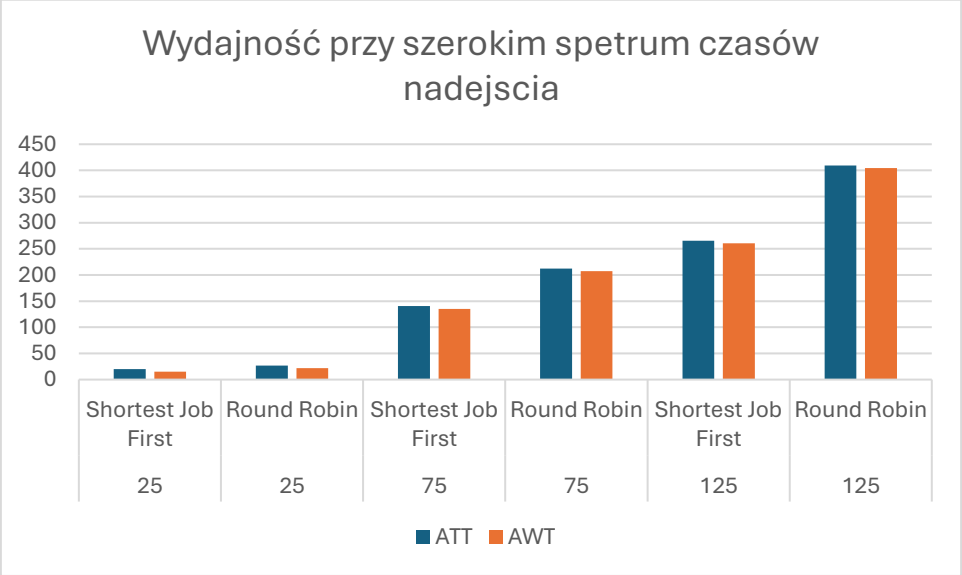
W końcowym etapie do zmiennej *results_cpu* zostają zwrócone średnie czasy działania dla każdego z algorytmów. Z wyników zostaje utworzony wykres. Oraz do osobnego pliku zostaną zapisane zgrupowane wyniki wszystkich testów.

```python
#Zapisanie skumulowanych wyników
with open("Merged.txt","w") as file:
    file.write("LiczbaProcesow;Algorytm;ATT;AWT\n")

for t in range(test_number):
    if os.path.exists(f"test{t}"):     # Dla każdego testu tworzymy osobny folder
        shutil.rmtree(f"test{t}")
    os.makedirs(f"test{t}", )
    os.chdir(f"test{t}")

    results_cpu = run_cpu_tests(settings_cpu,t,seed)  # Zwrot wyników testu
    print(results_cpu)
    makegraph_cpu(results_cpu,t) # Stworzenie wykresu dla pary algorytmów
    os.chdir("..")

    with open("Merged.txt", "a") as file:
        for key, value in results_cpu.items():
            file.write(f"{settings_cpu["Number Of Processes"][t]};{key};{value[0]};{value[1]}\n")
```

Rysunek 6

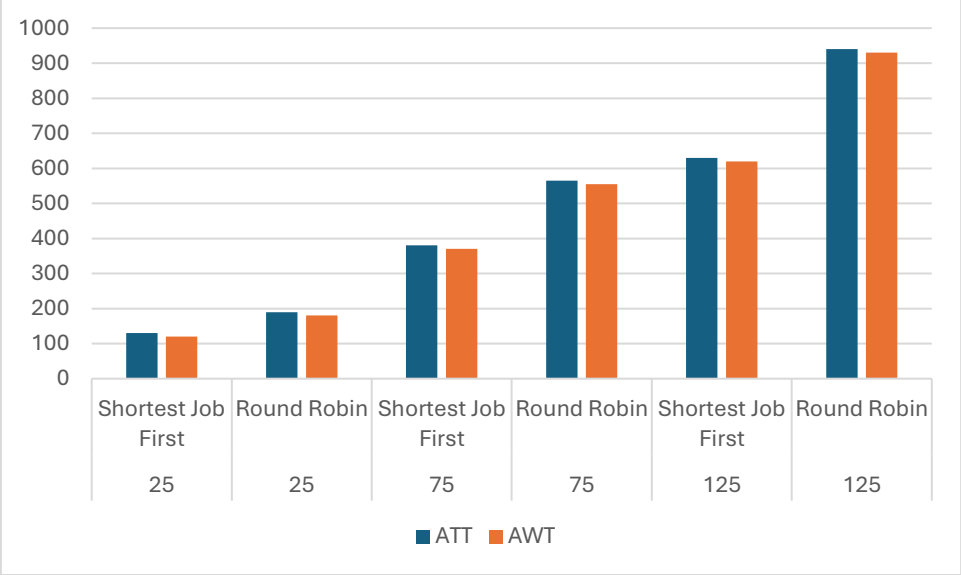
3. Przeprowadzone eksperymenty

a. 25,75,125 procesów, czasy nadejścia (0,100), czas wykonania 5, kwant czasu = 3



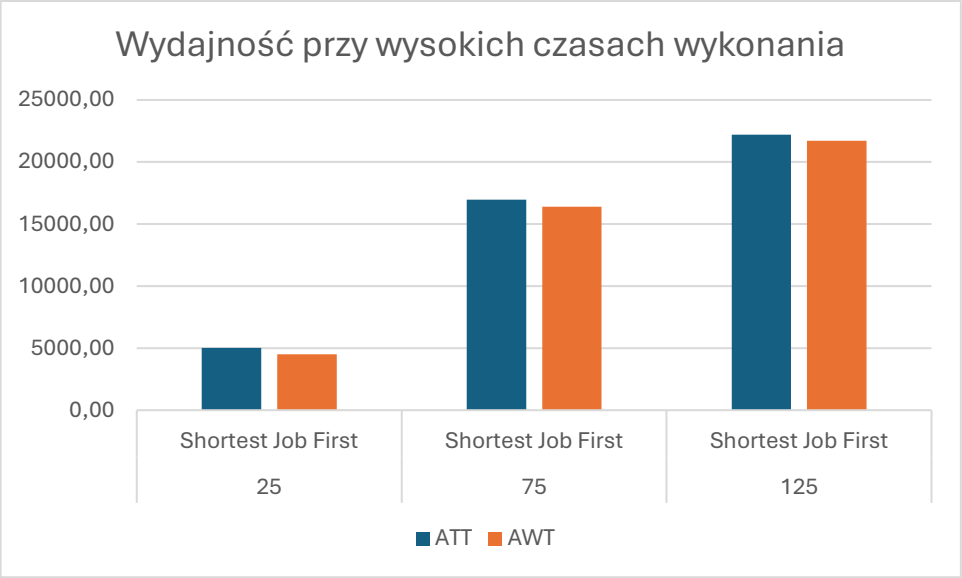
Rysunek 7

b. 25,75,125 procesów, czasy nadejścia 0,25,50, czas wykonania = 10, kwant czasu = 5



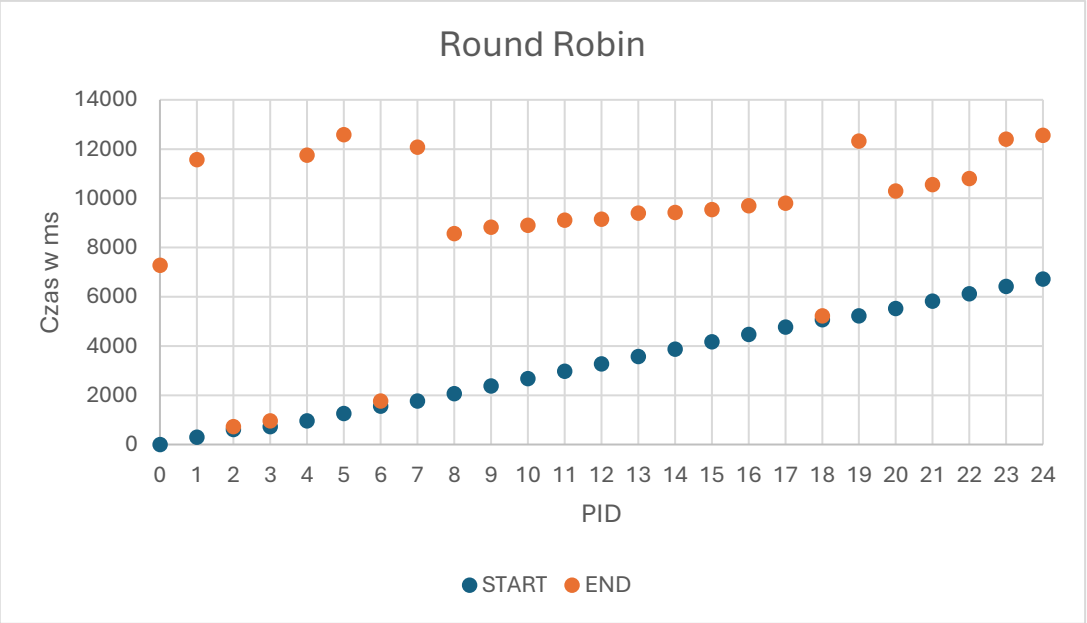
Rysunek 8

c. 25,75,125 procesów, czas nadejścia 0, czasy wykonania (100,1000), kwant czasu = 300

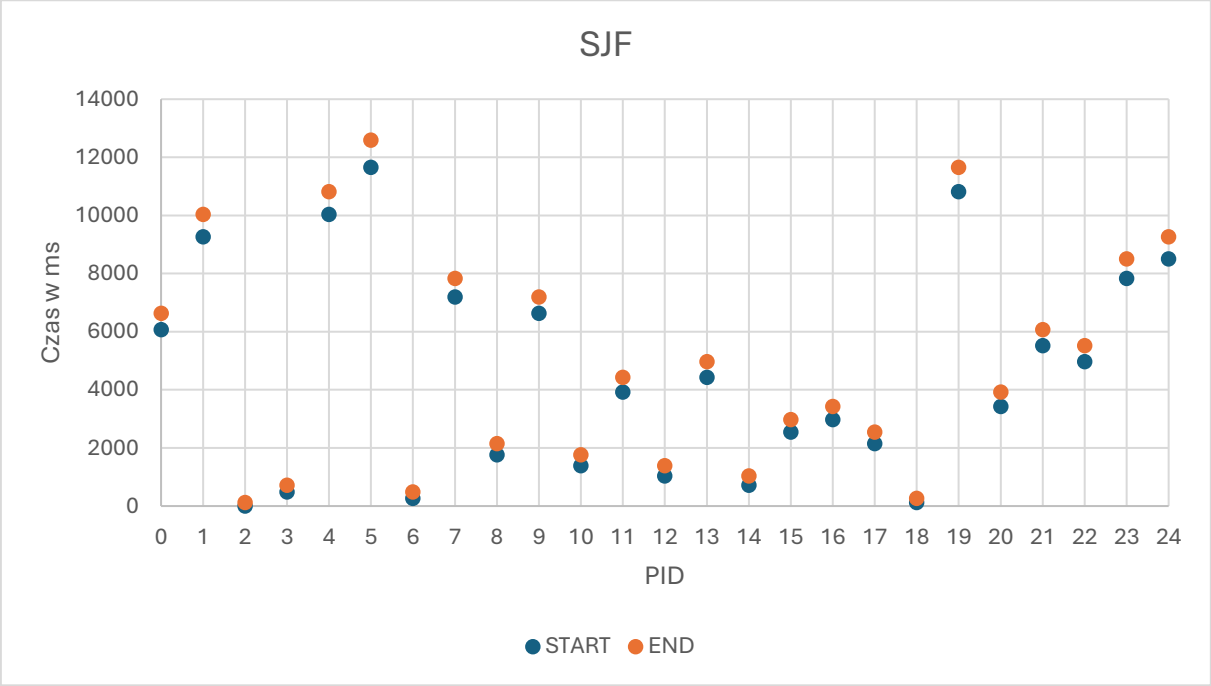


Rysunek 9

d. Różnica czasu od wejścia procesu do jego wykonania. 25 stron, czas przyjścia 0



Rysunek 10



Rysunek 11

4. Wnioski

Na podstawie wykresów możemy stwierdzić, że istnieje ogromna przepaść wydajnościowa pomiędzy algorytmami, która zwiększa się wraz

z ilością stron. Duże czasy oczekiwania i realizacji dla Round Robin wynikają z dużej ilości przełączeń kontekstu, które algorytm musi przeprowadzać po minięciu danego kwantu czasu.

Round Robin sprawdzi się w systemach interaktywnych, gdzie jest pożądana sprawiedliwość podziału procesów

Shortest Job First sprawdzi się w systemach wymagających minimalnego średniego czasu oczekiwania oraz, w których można przewidzieć czasy wykonania.

Symulacja zastępowania stron

1. Opis algorytmów FIFO, LRU, LFU

FIFO – zastępuje „najstarszą” stronę w pamięci.

LRU – Zastępuje najdawniej użytą stronę.

LFU – zastępuje najrzadziej używaną stronę.

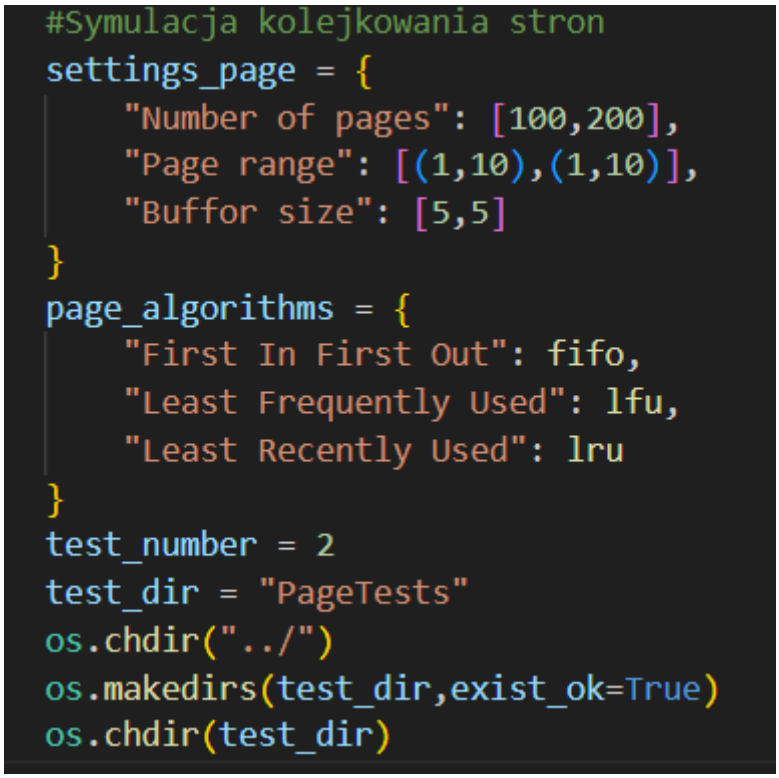
2. Opis procedury testowania

Strony są generowane za pomocą funkcji *generate_pages()*. (Rysunek 10.)

```
def generate_pages(num,p_range=(1,10),seed=None):
    random.seed(seed)
    return [random.randint(p_range[0],p_range[1]) for i in range(num) ]
```

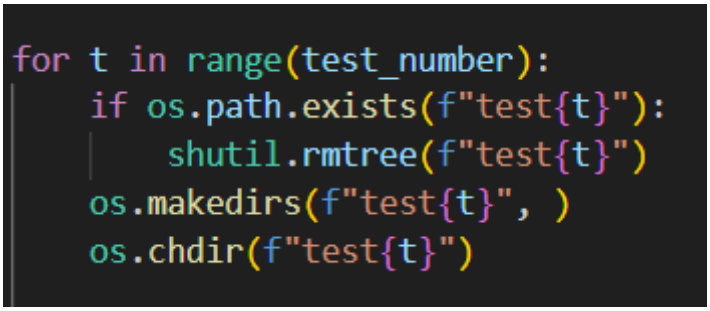
Rysunek 12

W głównym module możemy ustawić ustawienia do generowania stron. Ustalamy również folder, w którym zapiszemy nasze wyniki.(Rysunek 11.)



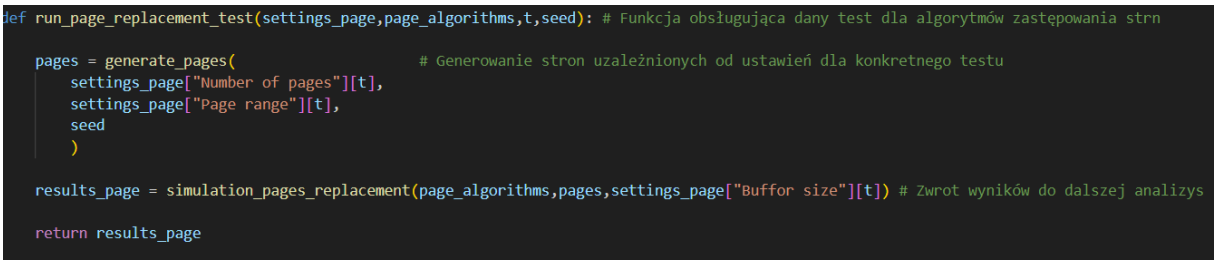
Rysunek 13

Dla każdego testu tworzymy strukturę plików.(Rysunek 12.)



Rysunek 14

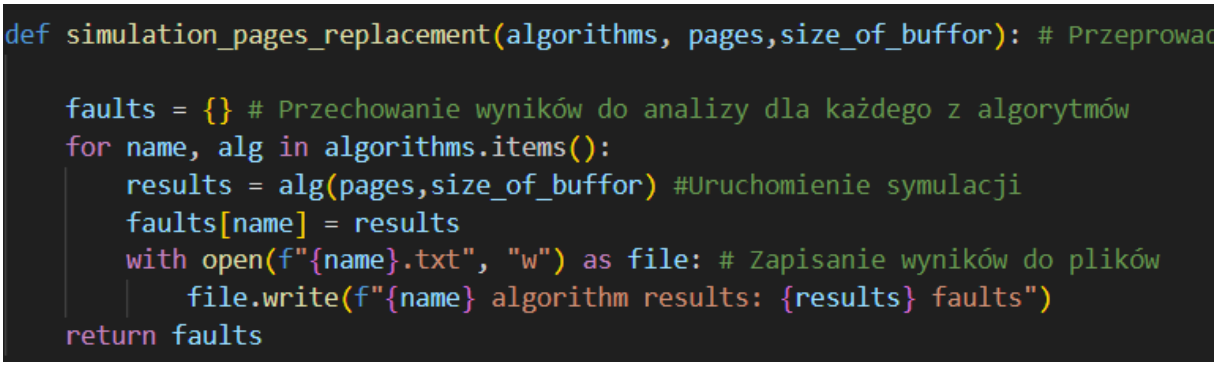
Każdy test zostaje uruchomiony funkcją *run_page_replacement_test()*. Są w niej generowane strony, które następnie są przekazywane do funkcji odpowiadającej za symulacje.(Rysunek 13.)



```
def run_page_replacement_test(settings_page,page_algorithms,t,seed): # Funkcja obsługująca dany test dla algorytmów zastępowania strn

    pages = generate_pages(                    # Generowanie stron uzależnionych od ustawień dla konkretnego testu
        settings_page["Number of pages"][t],
        settings_page["Page range"][t],
        seed
        )

    results_page = simulation_pages_replacement(page_algorithms,pages,settings_page["Buffor size"][t]) # Zwrot wyników do dalszej analizys

    return results_page
```

Rysunek 15

Funkcja *simulation_pages_replacement()* uruchamia każdy z algorytmów dla każdego testu. Zapisuje liczbę błędów do pliku. Zwraca liczbę błędów dla poszczególnych algorytmów.



```
def simulation_pages_replacement(algorithms, pages,size_of_buffor): # Przeprowac

    faults = {} # Przechowanie wyników do analizy dla każdego z algorytmów
    for name, alg in algorithms.items():
        results = alg(pages,size_of_buffor) #Uruchomienie symulacji
        faults[name] = results
        with open(f"{name}.txt", "w") as file: # Zapisanie wyników do plików
            file.write(f"{name} algorithm results: {results} faults")
    return faults
```

Rysunek 16

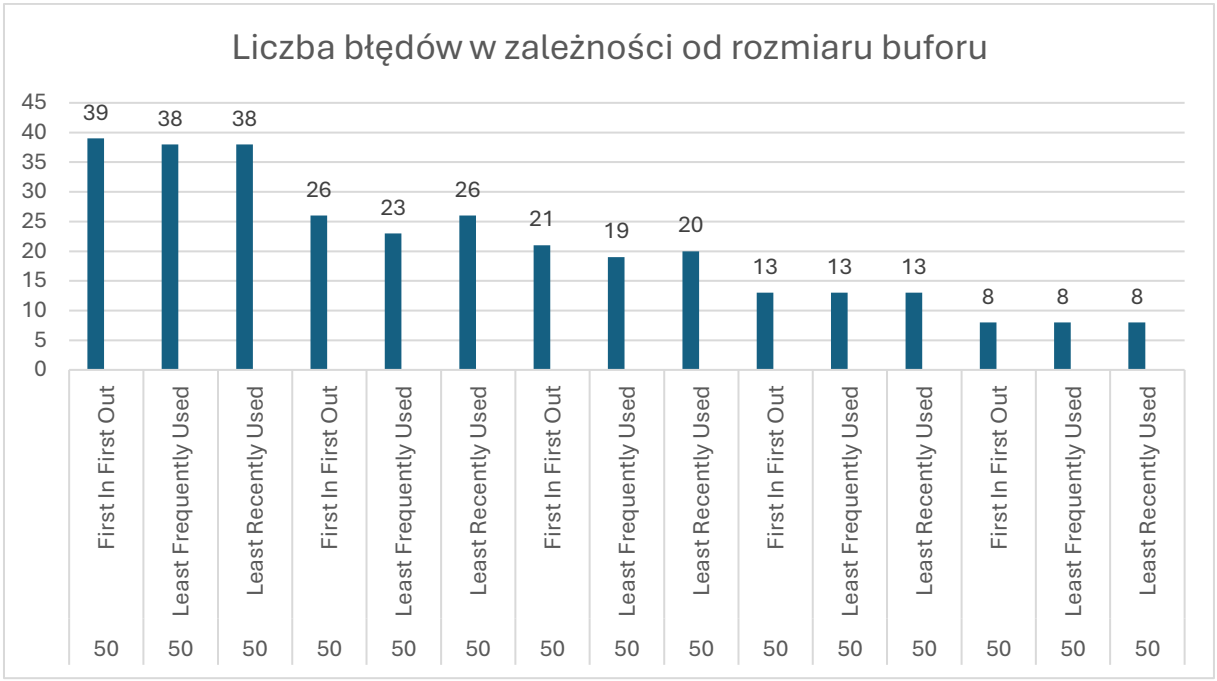
W końcowym z wyników zostają utworzone wykresy. Dodatkowe wyniki wszystkich testów są łączone w jednym pliku.(Rysunek 15)

```
with open("Merged.txt", "w") as file:
    file.write("LiczbaStron;Algorytm;PageFaults\n")

for t in range(test_number):
    if os.path.exists(f"test{t}"):
        shutil.rmtree(f"test{t}")
    os.makedirs(f"test{t}", )
    os.chdir(f"test{t}")

    results_page = run_page_replacement_test(settings_page,page_algorithms,t,seed) # Zwrot wyników testu
    makegraph_page(results_page,t) # Utworzenie wykresu dla algorytmów

    os.chdir("..")

    with open("Merged.txt", "a") as file:
        for key, value in results_page.items():
            file.write(f"{settings_page['Number of pages'][t]};{key};{value}\n")
```

Rysunek 17

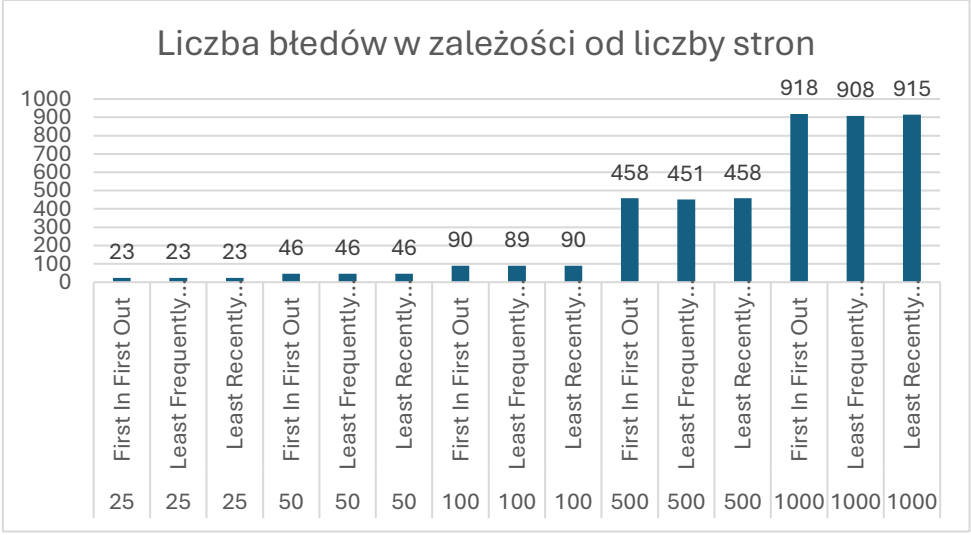
3. Eksperymenty

a. 50 stron, rozmiar pamięci 2,4,6,8,10, rozmiar stron (1,10)



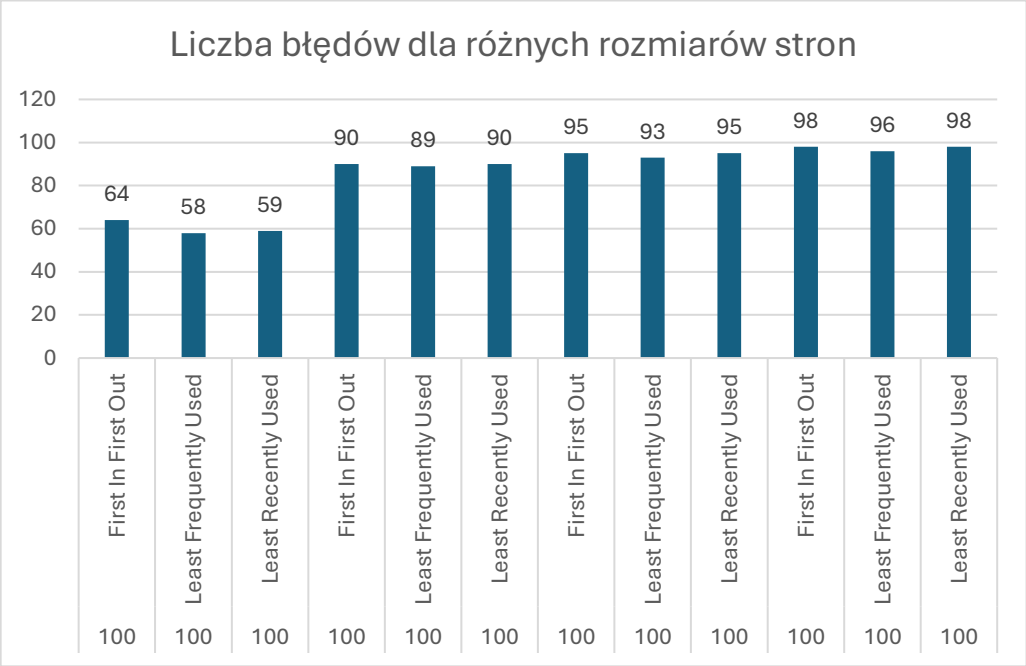
Rysunek 18

b. 25,50,100,500,1000 stron, bufor 4, zakres stron (1,50)



Rysunek 19

c. Liczba stron 100, zakres stron (1,10),(1,50),(1,100),(1,200), bufor 4



Rysunek 20

4. Wnioski

Najlepszą wydajnością charakteryzuje się LFU, co jest związane z tym, że często używane strony pozostają w pamięci. Algorytmy FIFO oraz LRU w niektórych przypadkach mogą prezentować podobną ilość *page faults*.