Sprawozdanie z programów symulacyjnych – Wiktor Łangowski

- I część sprawozdania będzie dotyczyła algorytmów przydziału procesora
- II część sprawozdania będzie dotyczyła algorytmów stronicowania

Algorytmy przydziału procesora

W tym sprawozdaniu omówię dwa algorytmy, których implementacje przedstawię poniżej. Są to algorytmy:

- FCFS
- SJF – nie wywłaszczający

Krótki opis działania algorytmów przydziału procesora

Algorytmy przydziału procesora zarządzają kolejnością wykonywania procesów w systemie operacyjnym. Celem tych algorytmów jest efektywne przydzielanie czasu procesora, zminimalizowanie czasu oczekiwania oraz poprawa ogólnej wydajności systemu. Różne algorytmy stosują odmienne strategie, aby osiągnąć te zamierzone cele. Ja opiszę wyłącznie analizowane przeze mnie algorytmy, czyli FCFS oraz SJF.

Algorytm FCFS

Jest to zdecydowanie jeden z najprostszych algorytmów przydziału procesora. Jak sama nazwa wskazuje First Come First Served, procesy są wykonywane w kolejności, w jakiej przychodzą do systemu, bez przerywania ich działania. Jest on stosunkowy prosty do implementacji, jednakże może on prowadzić do długiego czasu oczekiwania dla krótszych procesów, jeśli przed nimi są długie procesy. Ten proces jest nazywany „głodzeniem” procesów, które mogą długo czekać na przydział procesora.

Algorytm SJF

Shorted job first jest algorytmem, który przydziela procesor procesowi o najkrótszym czasie wykonania (burst time). Opisywanym przeze mnie rodzajem sjf jest ten niewywłaszczający, co oznacza, że proces nie jest przerywany po rozpoczęciu. Minimalizuje on średni czas oczekiwania (przeprowadzone zostanie porównanie czasów oczekiwania z algorytmem FCFS) co oczywiście pozytywnie wpływa na wydajność procesora oraz sprzyja on krótkim procesom – co za tym idzie może to prowadzić do głodzenia długich procesów przez ciągłe pojawianie się krótkich procesów.

Implementacja algorytmu FCFS oraz testowanie wykonanego przeze mnie kodu

Zaimplementowany przeze mnie kod algorytmu FCFS oblicza czasy oczekiwania oraz realizacji każdego procesu, a także oblicza średnie wartości tych czasów. Obliczone średnie wartości czasy oczekiwania posłużą później do porównania wydajności algorytmu FCFS oraz SJF

```python
def insert_processes():
    procesy = []
    liczba_procesów = int(input("Podaj liczbę procesów: "))
    for i in range(1, liczba_procesów + 1):
        burst_time = int(input(f"Podaj czas trwania procesu {i}: "))
        procesy.append({'numer_procesu': i, 'czas_trwania': burst_time})
    return procesy

procesy = insert_processes()
```

Rys. 1. Funkcja insert_processes()

Funkcja *insert_processes* pozwala użytkownikowi wprowadzić dane o procesach. Pobiera ona od użytkownika liczbę procesów, następnie pobiera czas trwania każdego procesu (burst_time) i dodaje go do listy jako słownik zawierający numer procesu oraz czas jego trwania.
Czas nadejścia procesu zależy od jego kolejności w liście: procesy są przetwarzane w kolejności w jakiej zostały dodane do listy. (Rys 2.)

```
Podaj liczbe procesow: 6
Podaj czas trwania procesu 1: 10
Podaj czas trwania procesu 2: 12
Podaj czas trwania procesu 3: 10
Podaj czas trwania procesu 4: 20
Podaj czas trwania procesu 5: 22
Podaj czas trwania procesu 6: 20
```

Rys 2. Zaimplementowanie rozwiązania nadejścia procesów wedle ich kolejności w liście.

```python
# Funkcja do obliczania czasu oczekiwania dla procesów
def waiting_time(procesy):
    czas_oczekiwania = [0] * len(procesy)  # Inicjalizacja listy czasów oczekiwania
    for i in range(1, len(procesy)):
        czas_oczekiwania[i] = czas_oczekiwania[i - 1] + procesy[i - 1]['czas_trwania']  # Obliczanie czasu oczekiwania
    return czas_oczekiwania

czasy_oczekiwania = waiting_time(procesy)

# Funkcja do obliczania czasu realizacji dla procesów
def realization_time(procesy, czasy_oczekiwania):
    czasy_realizacji = []
    for i, proces in enumerate(procesy):
        czas_realizacji = czasy_oczekiwania[i] + proces['czas_trwania']  # Obliczanie czasu realizacji
        czasy_realizacji.append(czas_realizacji)  # Dodanie czasu realizacji do listy
    return czasy_realizacji

czasy_realizacji = realization_time(procesy, czasy_oczekiwania)
```

Rys. 3. Dwie główne funkcje programu – obliczające czas oczekiwania i czas realizacji procesów (execution time)

W tej części chciałbym omówić w jaki sposób funkcje obliczają czas oczekiwania i czas realizacji. Pierwszym elementem było ustawienie pierwszej wartości czasu oczekiwania dla pierwszego przybyłego procesu = 0 (czas_oczekiwania[0] = 0). Następnie dla pozostałych procesów wzór był następujący:

czas_oczekiwania[i] = czas_oczekiwania[i – 1] + czas_trwania[i - 1] gdzie i to indeks procesu
w liście procesów.

Pozwoliło to uzyskać indywidualne czasy oczekiwania dla każdego pojedynczego procesu

Czas realizacji natomiast jest obliczany pojedynczym wzorem:

czas_realizacji[i] = czas_oczekiwania[i] + czas_twania[i] gdzie i to indeks procesu w liście procesów

```
Proces 1 ma czas oczekiwania : 0 i czas realizacji: 10
Proces 2 ma czas oczekiwania : 10 i czas realizacji: 22
Proces 3 ma czas oczekiwania : 22 i czas realizacji: 32
Proces 4 ma czas oczekiwania : 32 i czas realizacji: 52
Proces 5 ma czas oczekiwania : 52 i czas realizacji: 74
Proces 6 ma czas oczekiwania : 74 i czas realizacji: 94
```

Rys. 4. Rozpisanie czasów realizacji o oczekiwania dla wszystkich pojedynczych procesów

```python
# Funkcja do obliczania średniego czasu oczekiwania
def average_waiting_time(czasy_oczekiwania):
    suma_czasow = sum(czasy_oczekiwania)
    liczba_procesow = len(czasy_oczekiwania)
    sredni_czas_czekania = suma_czasow / liczba_procesow  # Obliczanie średniego czasu oczekiwania
    return sredni_czas_czekania

# Funkcja do obliczania średniego czasu realizacji
def average_realization_time(czasy_realizacji):
    suma_czasow = sum(czasy_realizacji)
    liczba_procesow = len(czasy_realizacji)
    sredni_czas_realizacji = suma_czasow / liczba_procesow  # Obliczanie średniego czasu realizacji
    return sredni_czas_realizacji


# Obliczanie i wyświetlanie średniego czasu oczekiwania
średni_czas_oczekiwania = average_waiting_time(czasy_oczekiwania)
print(f"Średni czas oczekiwania jest równy: {średni_czas_oczekiwania} sekund/y")

# Obliczanie i wyświetlanie średniego czasu realizacji
średni_czas_realizacji = average_realization_time(czasy_realizacji)
print(f"Średni czas realizacji jest równy: {średni_czas_realizacji} sekund/y")
```

Rys. 5. Obliczanie średnich czasów oczekiwania i realizacji

Ostatnią częścią tego kodu jest obliczenie średniej wartości czasu oczekiwania oraz średniego czasu realizacji procesów. Obliczona średnia wartość czasu oczekiwania pozwoli nam porównać wydajność algorytmu FCFS z algorytmem SJF.

```
Średni czas oczekiwania jest równy: 31.666666666666668 sekund/y
Średni czas realizacji jest równy: 47.333333333333336 sekund/y
```

Rys 6. Obliczony średni czas realizacji i oczekiwania

Implementacja algorytmu SJF oraz test wykonanego przeze mnie kodu

Zaimplementowany przeze mnie kod algorytmu SJF oblicza czasy oczekiwania oraz realizacji każdego procesu, a także oblicza średnie wartości tych czasów. Jest to wersja niewywłaszczająca tego algorytmu zatem nie przerywa ona wykonywania procesów.

```python
# Funkcja do wprowadzania danych o procesach
def insert_procces_data():
    procesy = []
    liczba_procesow = int(input("Podaj liczbe procesow: "))  # Pobranie liczby procesów od użytkownika

    # Pętla do wprowadzania czasu trwania dla każdego procesu
    for i in range(1, liczba_procesow + 1):
        burst_time = int(input(f"Podaj czas trwania procesu {i}: "))
        procesy.append({'numer_procesu': i, 'czas_trwania': burst_time })  # Dodanie procesu do listy

    return procesy

procesy = insert_procces_data()
```

Rys. 7. Funkcja insert_process_data

Funkcja *insert_processes_data* jest identyczna wobec funkcji *insert_processes* zastosowanej w algorytmie fcfs, pozwala ona użytkownikowi wprowadzać dane o procesach. **W przpadku nadejścia procesów o takim samym czasie wykonywania** (ponieważ algorytm sjf sortuje procesy względem najmniejszego czasu wykonywania) zawsze proces, który został jako pierwszy wprowadzony do listy (czyli zgodnie z wcześniejszym założeniem, że kolejność w liście określa czasy przybycia procesów) zostanie jako pierwszy wprowadzony do kolejki. Oto przykład

```
Podaj liczbe procesow: 5
Podaj czas trwania procesu 1: 4
Podaj czas trwania procesu 2: 2
Podaj czas trwania procesu 3: 6
Podaj czas trwania procesu 4: 2
Podaj czas trwania procesu 5: 7
Proces 2 ma czas oczekiwania : 0 i czas realizacji: 2
Proces 4 ma czas oczekiwania : 2 i czas realizacji: 4
Proces 1 ma czas oczekiwania : 4 i czas realizacji: 8
Proces 3 ma czas oczekiwania : 8 i czas realizacji: 14
Proces 5 ma czas oczekiwania : 14 i czas realizacji: 21
```

Rys. 8. Omówienie zaimplementowania odróżnienia czasu przybycia procesu

Na załączonym przykładzie można zauważyć, że proces 2 który ma taki sam wykonania jak proces 4 został wykonany jako pierwszy.

```python
# Funkcja pomocnicza do sortowania procesów po czasie trwania
def sort_key(proces):
    return proces['czas_trwania']

# Funkcja do sortowania procesów według czasu trwania
def sort_proccesses(procesy):
    procesy.sort(key=sort_key)
    return procesy

procesy_posortowane = sort_proccesses(procesy)
```

Rys 9. Dwie funkcje, które pozwalają nam uzyskać przeprowadzenie algorytmu SJF, czyli funkcje sortujące względem najmniejszego czasu trwania

Następne linijki kodu omówię pokrótce, ponieważ są identyczne względem implementacji w algorytmie fcfs (obliczanie czasu oczekiwania i realizacji oraz średniej tych czasów)

```python
# Funkcja do obliczania czasu oczekiwania dla posortowanych procesów
def waiting_time(procesy_posortowane):
    czasy_oczekiwania = [0] * len(procesy_posortowane)  # Inicjalizacja listy czasów oczekiwania
    for i in range(1, len(procesy_posortowane)):
        czasy_oczekiwania[i] = czasy_oczekiwania[i - 1] + procesy_posortowane[i - 1]['czas_trwania']  # Obliczanie czasu oczekiwania
    return czasy_oczekiwania

czasy_oczekiwania = waiting_time(procesy_posortowane)

# Funkcja do obliczania czasu realizacji dla posortowanych procesów
def exec_time(procesy_posortowane, czasy_oczekiwania):
    czasy_wykonania = []
    for i in range(len(procesy_posortowane)):
        czas_realizacji = czasy_oczekiwania[i] + procesy_posortowane[i]['czas_trwania']  # Obliczanie czasu realizacji
        czasy_wykonania.append(czas_realizacji)  # Dodanie czasu realizacji do listy
    return czasy_wykonania

czasy_wykonania = exec_time(procesy_posortowane, czasy_oczekiwania)

# Wyświetlanie wyników dla każdego procesu
for i, proces in enumerate(procesy_posortowane):
    print(f"Proces {proces['numer_procesu']} ma czas oczekiwania : {czasy_oczekiwania[i]} i czas realizacji: {czasy_wykonania[i]}")
```

Rys. 10. Obliczanie czasu oczekiwania i realizacji dla algorytmu SJF

```python
# Funkcja do obliczania średniego czasu oczekiwania
def average_waiting_time(czasy_oczekiwania):
    sredni_czas_oczekiwania = sum(czasy_oczekiwania) / len(czasy_oczekiwania)  # Obliczanie średniego czasu oczekiwania
    return sredni_czas_oczekiwania

sredni_czas_oczekiwania = average_waiting_time(czasy_oczekiwania)

# Funkcja do obliczania średniego czasu realizacji
def average_realization_time(czasy_wykonania):
    sredni_czas_wykonania = sum(czasy_wykonania) / len(czasy_wykonania)  # Obliczanie średniego czasu realizacji
    return sredni_czas_wykonania

sredni_czas_realizacji = average_realization_time(czasy_wykonania)

# Wyświetlanie wyników dla każdego procesu
for i, proces in enumerate(procesy_posortowane):
    print(f"Proces {proces['numer_procesu']} ma czas oczekiwania : {czasy_oczekiwania[i]} i czas realizacji: {czasy_wykonania[i]}")

# Wyświetlanie średniego czasu oczekiwania
print("Średni czas oczekiwania:", sredni_czas_oczekiwania)

# Wyświetlanie średniego czasu realizacji
print("Średni czas wykonania:", sredni_czas_realizacji)
```

Rys. 11. Obliczanie średnich czasów oczekiwania i realizacji dla algorytmu SJF

Omówienie części testowej oraz wyników eksperymentów porównywania algorytmów SJF vs FCFS

```
I część testów - parametry dla algorytmów SJF oraz FCFS
Będą to procesy o różnych wartosciach i różnych długościach.
1. [2,5,7,23,13,2]
2. [30,24,75,2,5,10]
3. [5,2,10,1,3,5,6]
4. [10,2,5,6,1,20,5,1]
5. [2,12,4,2,52
6. [5,3,6,2,4,6,7,2,3,6]
7. [10,5,2]
8. [3,2,9,21,12]
9. [18,2,33,10,9,3,4,6]
10.[1,3,5,7,9]
```

Rys. 12 Parametry dla algorytmów SJF oraz FCFS

W tej części sprawozdanie zostaną omówione wyniki czasów oczekiwania dla algorytmów przydziału procesora. Wykonałem 10 prób pomiarowych, aby wynik przeprowadzonego eksperymentu był wiarygodny oraz miarodajny. Czasy trwania procesów (czyli dane wpisane do tabeli) były dobierane losowo, jedynie 10 podpunkt był dobrany specjalnie, aby udowodnić co definiuje różnicę w czasie oczekiwania dla algorytmów. Dane były przeze mnie manualnie dodawane, poprzez wpisywania ich do funkcji insert_procesess (FCFS) oraz insert_process_data (SJF). **Kolejność procesów w liście określa ich czas nadejścia, zgodnie z tłumaczeniem na początku sprawozdania. Przykładowo w podpunkcie 7 są trzy procesy o czasach trwania (burst_time): 10, 5, 2. W algorytmie SJF przykładowo dla podpunktu pierwszego, jako pierwszy zostanie wykonany proces 1 mimo że ma taki sam czas trwania jak ostatni proces.**

Wyniki eksperymentu

Numer testu	Średni czas oczekiwania FCFS	Średni czas oczekiwania SJF
1	18,33	10
2	80,00	23,00
3	13,43	8,43
4	22,38	10,00
5	10,80	6,80
6	19,50	14,90
7	8,34	3,00
8	11,40	9,40
9	47,50	17,62
10	6,00	6,00

Rys. 13 Obliczone średnie czasy oczekiwania wobec parametrów testowych

- Z obserwacji wyników śmiało możemy stwierdzić, że algorytm SJF generalnie przewyższa algorytm FCFS w większości testów, co wynika z jego mechaniki przydzielania procesora procesom o najkrótszym czasie wykonania. Dzięki temu krótsze procesy są obsługiwane szybciej, co zapobiega im głodzeniu i zmniejsza ogólny czas oczekiwania. Algorytm FCFS natomiast przetwarza procesy w kolejności ich przybycia, co może prowadzić do długich czasów oczekiwania, zwłaszcza gdy długie procesy przychodzą przed krótkimi. To zjawisko jest szczególnie widoczne w testach takich jak 2 i 9, gdzie różnice w średnich czasach oczekiwania są znaczne.
- Chciałbym też zwrócić uwagę na test 10, gdzie oba algorytmy mają ten sam czas oczekiwania. Taka sytuacja jest tylko możliwa w przypadku, gdy procesy mają ten sam czas trwania lub nie zachodzi sortowania względem najkrótszego działania procesu. Taka sytuacja tutaj nastąpiła zgodnie z parametrami punktu 10, oba algorytmy działają w takim przypadku identycznie.
- Chciałbym też wytłumaczyć skąd wynika ta cała różnica w czasach oczekiwania na procesy. Fakt, że SJF ma tak krótki czas oczekiwania wynika wyłącznie z tego, że długie procesy są wykonywane na końcu. Oznacza to, że ich czas oczekiwania nie wpływa tak bardzo na średnią, ponieważ procesy te nie blokują krótszych procesów. W FCFS długie procesy na początku powodują, że wszystkie inny procesy muszą dłużej oczekiwać na swoją realizację.

Oto graficzne wytłumaczenie kolejności wykonywania procesów dla tych dwóch algorytmów. Wykonamy to dla podpunktu 7, czyli [10, 5 ,2] z części testowej.

Algorytm FCFS

Rys. 14. Graficzne rozrysowanie czasów oczekiwania dla procesów o czasie trwania [10, 5, 2] - FCFS

Średni czas oczekiwania w tym przypadku będzie wynosić (0 + 10 + 15) / 3= 8,33

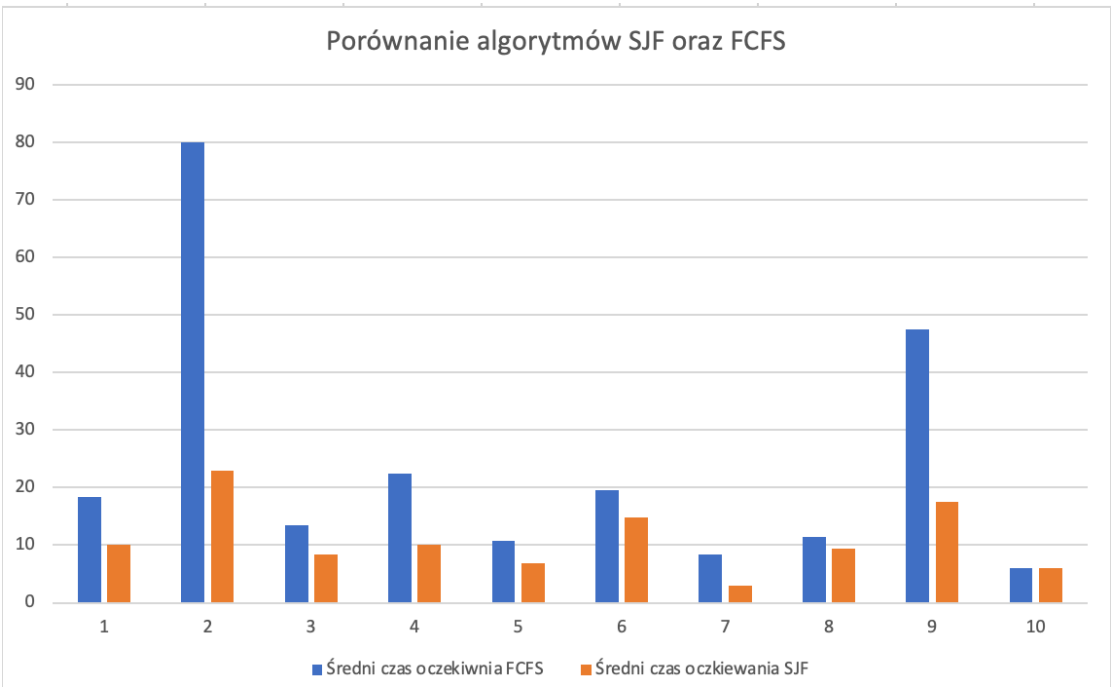
Algorytm SJF

Rys. 15. Graficzne rozrysowanie czasów oczekiwania dla procesów o czasie trwania [10, 5, 2] – SJF

Średni czas oczekiwania w tym przypadku będzie wynosić (0 + 2 + 7) / 3 = 3

Przypominam, że posługujemy się tym wzorem:

czas_oczekiwania[i] = czas_oczekiwania[i – 1] + czas_trwania[i - 1] , *został on objaśniony w początkowej części sprawozdanie*



Rys. 16 Wykres przedstawiający różnice w średnim czasie oczekiwania dla algorytmu FCFS oraz SJF

II cześć sprawozdania

Algorytmy stronicowania

W tej części sprawozdania omówię dwa algorytmy, których implementacje przedstawię poniżej. Są to algorytmy:

- FIFO
- LRU

Krótki opis algorytmów stronicowania

Algorytmy stronicowania są stosowane w systemach operacyjnych do zarządzania pamięcią wirtualną. Pamięć wirtualna pozwala na uruchamianie programów, które wymagają więcej

pamięci, niż jest fizycznie dostępne. Stronicowanie dzieli pamięć na małe bloki zwane stronami, które mogą być przenoszone między pamięcią główną a pamięcią masową. Gdy proces potrzebuje dostępu do strony, która nie jest aktualnie w pamięci głównej (tzw. "page fault"), system operacyjny musi wybrać stronę do usunięcia z pamięci, aby zwolnić miejsce na nową stronę. Algorytmy stronicowania określają, która strona powinna zostać usunięta. Ja przedstawię działanie dwóch wybranych przeze mnie algorytmów, czyli FIFO oraz LRU

Algorytm FIFO (First-in, First-Out)

Algorytm Fifo usuwa stronę, które najdłużej znajduje się w pamięci. Nowa strona jest dodawana na końcu kolejki, a strona z początku kolejki jest usuwana. Jest on prosty do zaimplementowania i prosty do zrozumienia. Problemem jednak tego algorytmu jest jego prostota, nie bierze on pod uwagę rzeczywistej częstotliwości ani wzorca dostępu do stron.

Algorytm LRU (Least Recently Used)

Algorytm LRU usuwa stronę, która najdłużej nie była używana. Zakłada, że strony, które były używane niedawno, będą używana ponownie w przyszłości, a te, które były nieużywane przez dłuższy czas, są mniej prawdopodobne do ponownego użycia. Jest on bardziej efektywny niż Fifo, ponieważ lepiej odzwierciedla rzeczywiste wzorce dostępu do stron, powinien prowadzić do zmniejszenia liczby page faults. Jednakże jest on trudniejszy do implementacji i wymaga dodatkowej pamięci i obliczeń do utrzymania informacji o czasie użycia stron.

Implementacja algorytmu FIFO oraz testowanie wykonanego przeze mnie kodu

```python
import random

# Funkcja implementująca algorytm zastępowania stron FIFO
def fifo_page_replacement(pages, frame_size):
    memory_frames = []
    page_faults = 0  # Licznik błędów stron

    # Przechodzenie przez każdą stronę w liście
    for page in pages:
        if page not in memory_frames:  # Sprawdzenie, czy strona nie jest już w liście ramek pamięci
            if len(memory_frames) == frame_size:  # Sprawdzenie, czy ramki są pełne
                memory_frames.pop(0)  # Usunięcie najstarszej strony (FIFO)
            memory_frames.append(page)  # Dodanie nowej strony do listy ramek pamięci
            page_faults += 1  # Zwiększenie licznika błędów stron
            print(f"Strona {page} została dodana, ramki: {memory_frames}")
        else:
            print(f"Strona {page} jest juz w ramce: {memory_frames}")

    return page_faults  # Zwrócenie liczby błędów stron

frame_size = 3  # Rozmiar ramki pamięci

num_pages = 20  # Liczba stron do wygenerowania

# Wygenerowanie losowego ciągu stron
pages = [random.randint(0, 6) for i in range(num_pages)]

print("Wygenerowane strony:", pages)

# Wywołanie funkcji zastępowania stron FIFO i zapisanie liczby błędów stron
faults = fifo_page_replacement(pages, frame_size)

print("Liczba błędów stron: ", faults)  # Wyświetlenie liczby błędów stron
```

Rys. 17 Implementacja algorytmu FIFO w pythonie

Tutaj w odróżnieniu do trudniejszych algorytmów przydziału procesora całość algorytmu udało się zmieścić w jednej funkcji – *fifo_page_replacement*. Funkcja zwraca ważną dla nas wartość błędów strony (page_faults), które następnie będzie porównywana z wynikiem dla algorytmu LRU. Dodałem również wyświetlanie stron, aby użytkownik interaktywnie mógł obserwować proces działania algorytmu FIFO. Użyłem wyrażeń listowych takich jak pop i append, która odpowiednio usuwają najstarszy element z listy i dodają nowy element na początek listy. Pokazywana jest również lista błędów stron, które jak pisałem wcześniej będzie niezbędna w celu porównania algorytmów. Dodałem również losowe generowanie się liczb w liście *pages* aby wynik, aby w etapie testowania również porównać jak te algorytmy działają w przypadku wykonaniu wielu prób przy losowo generowanych liczbach.

```
Wygenerowane strony: [5, 3, 1, 5, 6, 6, 4, 6, 2, 3]
Strona 5 została dodana, ramki: [5]
Strona 3 została dodana, ramki: [5, 3]
Strona 1 została dodana, ramki: [5, 3, 1]
Strona 5 jest juz w ramce: [5, 3, 1]
Strona 6 została dodana, ramki: [3, 1, 6]
Strona 6 jest juz w ramce: [3, 1, 6]
Strona 4 została dodana, ramki: [1, 6, 4]
Strona 6 jest juz w ramce: [1, 6, 4]
Strona 2 została dodana, ramki: [6, 4, 2]
Strona 3 została dodana, ramki: [4, 2, 3]
Liczba błędów stron:  7
```

Rys. 18 Wynik wykonanego przeze mnie kodu: pokazanie wygenerowanych losowo liczb, pokazanie w jaki sposób strony są dodawane do listy oraz pokazanie liczby błędów stron.

Implementacja algorytmu LRU oraz testowanie wykonanego przeze mnie kodu

```python
import random

# Funkcja implementująca algorytm zastępowania stron LRU (Least Recently Used)
def lru_page_replacement(pages, frame_size):
    memory_frames = []  # Lista reprezentująca ramki pamięci
    last_used = []  # Lista przechowująca indeksy użycia stron
    page_faults = 0  # Licznik błędów stron

    # Przejście przez każdą stronę w liście
    for i, page in enumerate(pages):
        if page in memory_frames:  # Sprawdzenie, czy strona jest już w liście ramek
            last_used_index = memory_frames.index(page)  # Pobranie indeksu strony
            last_used[last_used_index] = i  # Aktualizacja indeksu ostatniego użycia strony
            print(f"Strona {page} została odświeżona, ramki: {memory_frames}")
```

Rys. 19 Początek funkcji lru_page_replacement – dodanie dodatkowej listy względem algorytmu fifo i dodanie nowej mechaniki

```
        else:
            if len(memory_frames) < frame_size:  # Sprawdzenie, czy są wolne ramki
                memory_frames.append(page)  # Dodanie nowej strony do listy ramek
                last_used.append(i)  # Dodanie indeksu użycia nowej strony
                print(f"Strona {page} została dodana, ramki: {memory_frames}")
            else:
                oldest_page_index = last_used.index(min(last_used))  # Znalezienie strony najdawniej używanej
                removed_page = memory_frames.pop(oldest_page_index)  # Usunięcie najdawniej używanej strony z listy ramek
                last_used.pop(oldest_page_index)  # Usunięcie indeksu najdawniej używanej strony
                memory_frames.append(page)  # Dodanie nowej strony do ramek
                last_used.append(i)  # Dodanie indeksu użycia nowej strony
                print(f"Usunięto stronę {removed_page}, a dodano stronę {page}, ramki: {memory_frames}")
            page_faults += 1  # Zwiększenie licznika błędów stron

    return page_faults  # Zwrócenie liczby błędów stron

frame_size = 3  # Rozmiar ramek pamięci

num_pages = 20 # Liczba stron do wygenerowania

# Wygenerowanie losowego ciągu stron
pages = [random.randint(0, 6) for i in range(num_pages)]
print("Wygenerowane strony:", pages)

# Wywołanie funkcji zastępowania stron LRU i zapisanie liczby błędów stron
faults = lru_page_replacement(pages, frame_size)
# Wyświetlenie liczby błędów stron
print("Liczba błędów stron: ", faults)
```

Rys. 20. Dalsza mechnika działania algorytmu LRU

Tak jak pisałem powyżej implementacja algorytmu LRU jest bardziej skomplikowana i wymaga dodatkowych linijek kodu w celu śledzenia najdawniej używanej strony. W tym przypadku to zadanie jest przeznaczone dla listy last_used, która przechowuje indeksy użycia stron. W przypadku wystąpienia danej strony ponownie indeks tej strony jest aktualizowany, czyli w przypadku gdy wcześniej dany indeks wynosił = 2, w przypadku ponownego wystąpienia strony zostanie zaktualizowany względem wartości „i” w pętli. Dalsze wytłumaczenie tej mechaniki widoczne jest na Rys. 20, gdzie możemy zauważyć, że strona najdawniej używana to ta z najmniejszym indeksem i właśnie ona jest usuwana. Dodam jeszcze, że ta lista jest aktualizowana zgodnie z iteracją po pętli za pomocą wartości „i”.

Tutaj również zaimplementowałem mechanikę losowego generowania liczb która wystąpiła już w algorytmie fifo. Wyświetlane są również wygenerowane strony, liczbę błędów stron oraz mechanika działania algorytmu lru, żeby była ona czytelna dla użytkownika.

```
Wygenerowane strony: [6, 6, 2, 1, 5, 0, 3, 1, 6, 3]
Strona 6 została dodana, ramki: [6]
Strona 6 została odświeżona, ramki: [6]
Strona 2 została dodana, ramki: [6, 2]
Strona 1 została dodana, ramki: [6, 2, 1]
Usunięto stronę 6, a dodano stronę 5, ramki: [2, 1, 5]
Usunięto stronę 2, a dodano stronę 0, ramki: [1, 5, 0]
Usunięto stronę 1, a dodano stronę 3, ramki: [5, 0, 3]
Usunięto stronę 5, a dodano stronę 1, ramki: [0, 3, 1]
Usunięto stronę 0, a dodano stronę 6, ramki: [3, 1, 6]
Strona 3 została odświeżona, ramki: [3, 1, 6]
Liczba błędów stron:  8
```

Rys. 21. Wynik wykonanego przeze mnie kodu: pokazanie wygenerowanych losowo liczb, pokazanie w jaki sposób strony są dodawane do listy oraz pokazanie liczby błędów stron.

Omówienie części testowej oraz wyników eksperymentów porównywania algorytmów FIFO vs LRU

Test rozpoczniemy od porównania 15 testów wygenerowanych losowo wyników dla algorytmu FIFO oraz LRU. Przyjmijmy, że maksymalna wartość ramki pamięci wynosi = 3 oraz liczby stron = 20

```
Wartości wygenerowane losowo (LRU)

1.[2, 0, 1, 3, 3, 6, 2, 2, 5, 3, 5, 1, 4, 5, 5, 3, 3, 3, 4, 1]
2.[1, 6, 2, 3, 5, 2, 2, 0, 2, 6, 2, 1, 1, 3, 2, 2, 5, 1, 4, 6]
3.[2, 0, 6, 2, 4, 3, 0, 1, 4, 4, 4, 1, 4, 1, 3, 6, 4, 4, 3, 4]
4.[2, 6, 3, 5, 0, 6, 0, 2, 6, 6, 5, 5, 6, 1, 6, 4, 4, 5, 0, 4]
5.[0, 3, 0, 1, 0, 1, 5, 5, 3, 1, 1, 0, 1, 6, 4, 1, 4, 3, 0, 0]
6.[1, 4, 1, 3, 6, 6, 3, 2, 0, 5, 1, 6, 4, 0, 4, 2, 4, 4, 0, 5]
7.[0, 0, 6, 4, 2, 1, 2, 0, 6, 1, 2, 2, 4, 2, 1, 1, 5, 4, 1, 1]
8.[1, 3, 6, 6, 6, 2, 2, 2, 3, 4, 2, 5, 1, 5, 1, 0, 1, 5, 0, 3]
9.[0, 4, 4, 0, 0, 5, 2, 5, 2, 2, 0, 1, 4, 2, 4, 5, 5, 5, 2, 3]
10.[1, 6, 2, 4, 0, 5, 2, 0, 3, 4, 0, 3, 2, 1, 3, 6, 0, 5, 2, 0]
11.[3, 6, 0, 4, 2, 1, 3, 2, 0, 1, 3, 6, 6, 2, 1, 5, 1, 5, 2, 4]
12.[5, 5, 5, 6, 4, 3, 1, 4, 4, 1, 3, 0, 4, 0, 0, 3, 5, 0, 4, 5]
13.[2, 1, 0, 4, 0, 5, 0, 4, 3, 5, 5, 6, 4, 0, 2, 2, 0, 3, 5, 0]
14.[3, 2, 3, 6, 6, 5, 5, 2, 1, 3, 5, 3, 0, 4, 6, 2, 3, 5, 0, 1]
15.[0, 4, 4, 6, 5, 3, 1, 6, 5, 0, 2, 1, 1, 2, 2, 6, 0, 6, 4, 3]

Wartości wygenerowane losowo (FIFO)
1.[1, 4, 6, 6, 3, 1, 6, 5, 1, 3, 2, 6, 2, 3, 4, 4, 6, 1, 3, 1]
2.[1, 5, 6, 6, 2, 1, 4, 3, 3, 3, 2, 0, 6, 5, 0, 2, 2, 0, 1, 6]
3.[0, 2, 0, 5, 0, 0, 3, 6, 6, 6, 3, 3, 3, 6, 0, 6, 2, 1, 6, 0]
4.[0, 3, 1, 6, 4, 6, 3, 4, 0, 1, 0, 0, 6, 2, 4, 3, 6, 2, 1, 6]
5.[5, 1, 3, 1, 1, 4, 6, 4, 5, 6, 0, 3, 2, 0, 1, 0, 4, 3, 5, 0]
6.[3, 4, 0, 6, 1, 3, 4, 2, 3, 3, 2, 5, 5, 1, 4, 3, 6, 2, 0, 3]
7.[1, 6, 3, 5, 1, 2, 3, 5, 1, 0, 0, 5, 1, 4, 6, 6, 4, 2, 1, 6]
8.[4, 6, 4, 4, 1, 1, 1, 5, 3, 1, 5, 1, 2, 3, 6, 5, 1, 3, 6, 1]
9.[4, 6, 6, 6, 3, 1, 4, 5, 3, 0, 5, 0, 4, 4, 6, 0, 1, 6, 0, 4]
10.[4, 2, 0, 2, 3, 3, 5, 2, 4, 6, 1, 5, 0, 4, 5, 1, 5, 1, 1, 1]
11.[4, 5, 2, 2, 6, 6, 6, 3, 1, 2, 5, 4, 0, 1, 0, 1, 0, 2, 6, 2]
12.[6, 5, 6, 2, 1, 4, 2, 6, 5, 6, 1, 1, 6, 3, 5, 0, 1, 4, 5, 6]
13.[1, 5, 0, 6, 5, 5, 0, 2, 3, 3, 3, 1, 6, 0, 4, 5, 6, 1, 4, 0]
14.[5, 1, 4, 6, 6, 4, 0, 0, 5, 0, 1, 2, 2, 4, 5, 6, 0, 3, 1, 0]
15.[1, 2, 5, 4, 5, 2, 2, 2, 5, 6, 6, 0, 2, 0, 5, 5, 5, 6, 4, 6]
```

Rys. 22 Wartości wygenerowane losowo poszczególnie dla FIFO oraz LRU

Numer testu	Page faults FIFO	Page faults LRU	Sredni wynik dla fifo	Średni wynik dla lru
1	11	12	13,27	12,27
2	15	13		
3	10	11		
4	15	12		
5	15	10		
6	16	13		
7	14	12		
8	11	9		
9	13	9		
10	14	15		
11	13	15		
12	13	9		
13	15	13		
14	14	16		
15	10	15		

Rys. 23 Wynik dla wartości wygenerowanych losowo

- Jak widać na podstawie średnich wyników dla fifo oraz lru, lru ma niższą średnią liczbę błędów stron niż Fifo. Oznacza to, że w większości przypadków LRU jest bardziej efektywny niż FIFO pod względem liczby błędów stron – zostanie to zaprezentowane na przykładach manualnych
- W niektórych przypadkach (testy 1,10, 15) LRU ma nieco więcej błędów stron niż FIFO, co może być wynikiem specyficznego wzorca dostępu do stron w tych testach – strony były losowo zatem założenie lru, że element najdawniej używany nie powinien zostać ponownie użyty w najbliższym czasie jest niekoniecznie trafne w tym przypadku.
- Algorytm Fifo usuwa strony w kolejności ich załadowania do pamięci, co może prowadzić do sytuacji, gdzie często używana strona jest usuwana, mimo że nadal jest potrzebna
- Algorytm LRU zwykle prowadzi do mniejszej liczby błędów stron, ponieważ lepiej odpowiada rzeczywistym wzorcom dostępu do pamięci
- Wyniki są tutaj zbliżone do siebie, ponieważ nastąpiło losowe generowanie stron. Tutaj możemy wyciągnąć wniosek, że w przypadku braku określonego wzorca – algorytmy stronicowania będą osiągały bardzo podobne wyniki.

Kolejny test chciałbym, żeby przedstawiał w jakich sytuacjach dane algorytmy będą osiągały lepsze / gorsze wyniki. Ten test będzie na wartościach manualnie przeze mnie wprowadzonych – zostały ode dobrane, aby odzwierciedlały typowym wzorcom dostępu do pamięci.

```
Wartości wpisane manualnie:

1.[1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]
2.[1, 2, 3, 1, 4, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4]
3.[4, 3, 2, 1, 4, 3, 2, 5, 4, 3, 2, 1, 6, 7, 8, 1]
4.[1, 2, 1, 3, 1, 4, 1, 5, 1, 6, 1, 7, 1, 8, 1, 9]
5.[5, 6, 7, 8, 5, 6, 7, 8, 5, 6, 7, 8, 1, 2, 3, 4]
```

Rys. 24 Dobrane przeze mnie manualne wartości do wspólnego porównania dla dwóch algorytmów

Numer testu	Page faults FIFO	Page Faults LRU	Średni wynik FIFO	Średni wynik LRU
1	9,00	10,00	13,40	13,00
2	15,00	14,00		
3	16,00	16,00		
4	11,00	9,00		
5	16,00	16,00		

Rys. 25 Wynik wprowadzonych przeze mnie wartości manualnych

Jak widać tutaj również średni wynik lru był niższy od średniego wyniku fifo. Mimo że różnica nie jest duża, sugeruje to, że w ogólnym przypadku LRU jest nieco bardziej efektywny w minimalizowaniu liczby błędów stron.

Pomimo, że różnica w średniej liczbie błędów stron jest niewielka, algorytm LRU generalnie oferuje lepszą wydajność w kontekście minimalizacji błędów stron. Jednakże, wyniki pokazują, że w niektórych przypadkach FIFO może być równie efektywny. Dlatego wybór algorytmu powinien uwzględniać specyficzne wzorce dostępu do stron w danym systemie.