Sprawozdanie - programy symulacyjne

Maja Stec

Do mojego sprawozdania wybrałam po dwa z dostępnych algorytmów. Dla symulacji planowania czasu procesora wybrałam algorytm **FCFS** oraz **SJF**, a dla symulacji zastępowania stron **FIFO** i **LRU**. Wszystkie algorytmy zostały napisane w języku Python.

Algorytmy planowania czasu procesora

Algorytm **FCFS** (First Come First Serve) - jeden z najprostszych algorytmów planowania czasu procesów, który decyduje o kolejności wykonania w zależności od czasu przybycia - proces, który pojawi się pierwszy zostanie obsłużony pierwszy.

Algorytm **SJF** (Shortest Job First) - algorytm cechujący się najkrótszym czasem oczekiwania, którego minusem może być odsyłanie dłuższych procesów na koniec kolejki, jeśli krótsze będą się cały czas pojawiać. Procesy z najkrótszym czasem wykonania oraz przybycia będą umieszczane na początku.

Algorytmy zastępowania stron

Algorytm **FIFO** (First In First Out) - jeden z prostszych algorytmów zastępowania stron, aby zwolnić miejsce usuwa najstarszą stronę z kolejki. Nie uwzględnia tego czy strona jest często używana czy nie - jeśli jest najstarsza, jest pierwsza w kolejce do wyrzucenia.

Algorytm **LRU** (Least Recently Used) - algorytm usuwa stronę, która najdłużej zostaje nieużywana, a następnie zastępuje ją nową stroną i aktualizuje dane o użyciu stron. Minimalizuje to ryzyko usunięcia strony często używanej, tak jak następuje to w przypadku FIFO.

ALGORYTMY PLANOWANIA CZASU PROCESORA

-- FCFS --

```python
#Wczytywanie liczby i parametrów procesów z pliku
procesy = []
with open("dane_procesy.txt", "r", encoding="utf-8") as plik:
    linie = [linia.strip() for linia in plik if linia.strip()]
    liczba_procesow = int(linie[0])
    for i in range(1, liczba_procesow + 1):
        czas_przybycia, czas_wykonania = map(int, linie[i].split())
        procesy.append({
            'proces_id': f'P{i}',
            'czas_przybycia': czas_przybycia,
            'czas_wykonania': czas_wykonania
        })


#Sortowanie według czasu przybycia
procesy.sort(key=lambda x: x['czas_przybycia'])

#Obliczanie czasu wyjścia, realizacji i oczekiwania
czas_początkowy = 0
for p in procesy:
    if czas_początkowy < p['czas_przybycia']:
        czas_początkowy = p['czas_przybycia']
    p['czas_wyjscia'] = czas_początkowy + p['czas_wykonania']
    czas_początkowy = p['czas_wyjscia']
    p['realizacja'] = p['czas_wyjscia'] - p['czas_przybycia']
    p['oczekiwanie'] = p['realizacja'] - p['czas_wykonania']

#Obliczanie średniego czasu oczekiwania
sredni_czas_oczekiwania = sum(p['oczekiwanie'] for p in procesy) / liczba_procesow

#Wyświetlanie wyników
print("\n{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}".format(
    "Proces", "Czas przybycia", "Czas wykonania", "Czas wyjścia", "Czas realizacji", "Czas oczekiwania"))

for p in procesy:
    print("{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}".format(
        p['proces_id'],
        p['czas_przybycia'],
        p['czas_wykonania'],
        p['czas_wyjscia'],
        p['realizacja'],
        p['oczekiwanie']
    ))

print(f"\nŚredni czas oczekiwania: {sredni_czas_oczekiwania:.2f}")

#Zapisywanie wyników do pliku
with open("wyniki_fcfs.txt", "w", encoding="utf-8") as wynik_plik:
    wynik_plik.write("FCFS - Wyniki:\n")
    wynik_plik.write("{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}\n".format(
        "Proces", "Czas przybycia", "Czas wykonania", "Czas wyjścia", "Czas realizacji", "Czas oczekiwania"))

    for p in procesy:
        wynik_plik.write("{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}\n".format(
            p['proces_id'], p['czas_przybycia'], p['czas_wykonania'],
            p['czas_wyjscia'], p['realizacja'], p['oczekiwanie']
        ))

    wynik_plik.write(f"\nŚredni czas oczekiwania: {sredni_czas_oczekiwania:.2f}\n")
```

Dane potrzebne do wykonania zadania zostały umieszczone w pliku o nazwie “dane_procesy.txt” w postaci:

3
0 5
1 8
2 3

gdzie pierwsza cyfra oznacza liczbę procesów, a w każdej kolejnej linijce znajduje się czas przybycia oraz czas wykonania procesu. Algorytm wczytuje te dane do skryptu, a po wykonaniu wyświetla wyniki oraz umieszcza je w osobnym pliku o nazwie “wyniki_fcfs.txt”

```
FCFS - Wyniki:
Proces   Czas przybycia  Czas wykonania  Czas wyjścia  Czas realizacji    Czas oczekiwania
P1       0               5               5             5                  0
P2       1               8               13            12                 4
P3       2               3               16            14                 11

Średni czas oczekiwania: 5.00
```

```
Proces   Czas przybycia  Czas wykonania  Czas wyjścia  Czas realizacji    Czas oczekiwania
P1       0               5               5             5                  0
P2       1               8               13            12                 4
P3       2               3               16            14                 11

Średni czas oczekiwania: 5.00

Process finished with exit code 0
```

Wyniki zostały sformatowane, aby ułatwić odczyt i zwiększyć przejrzystość.

WNIOSKI

- Algorytm FCFS jest prosty w użyciu i bardzo czytelny
- Przez to, że przyjmuje procesy po kolei możemy uniknąć sytuacji, że jakiś proces zostanie pominięty przez swoje parametry
- Przy większych procesach o wcześniejszym czasie przybycia czas oczekiwania mocno się zwiększa, co nie jest optymalnym rozwiązaniem
- Krótsze procesy zostają umieszczone na końcu, jeśli będą miały większy czas przybycia, więc mimo, że wykonanie ich zajęłoby mało czasu, muszą czekać na zakończenie większych procesów

-- SJF--

```python
#Wczytywanie liczby i parametrów procesów z pliku
procesy = []
with open("dane_procesy.txt", "r", encoding="utf-8") as plik:
    linie = [linia.strip() for linia in plik if linia.strip()]
    liczba_procesow = int(linie[0])
    for i in range(1, liczba_procesow + 1):
        czas_przybycia, czas_wykonania = map(int, linie[i].split())
        procesy.append({
            'proces_id': f'P{i}',
            'czas_przybycia': czas_przybycia,
            'czas_wykonania': czas_wykonania
        })

# Sortowanie według czasu wykonania
procesy.sort(key=lambda p: p["czas_wykonania"])

# Obliczanie czasu wyjścia, realizacji i oczekiwania
czas_początkowy = 0
for p in procesy:
    if czas_początkowy < p['czas_przybycia']:
        czas_początkowy = p['czas_przybycia']
    p['czas_wyjscia'] = czas_początkowy + p['czas_wykonania']
    czas_początkowy = p['czas_wyjscia']
    p['realizacja'] = p['czas_wyjscia'] - p['czas_przybycia']
    p['oczekiwanie'] = p['realizacja'] - p['czas_wykonania']

# Obliczanie średniego czasu oczekiwania
sredni_czas_oczekiwania = sum(p['oczekiwanie'] for p in procesy) / liczba_procesow

# Wyświetlanie wyników
print("\n{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}".format(
    "Proces", "Czas przybycia", "Czas wykonania", "Czas wyjścia", "Czas realizacji", "Czas oczekiwania"))

for p in procesy:
    print("{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}".format(
        p['proces_id'],
        p['czas_przybycia'],
        p['czas_wykonania'],
        p['czas_wyjscia'],
        p['realizacja'],
        p['oczekiwanie']
    ))


print(f"\nŚredni czas oczekiwania: {sredni_czas_oczekiwania:.2f}")

#Zapisywanie wyników do pliku

with open("wyniki_sjf.txt", "w", encoding="utf-8") as wynik_plik:
    wynik_plik.write("SJF - Wyniki:\n")
    wynik_plik.write("{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}\n".format(
        "Proces", "Czas przybycia", "Czas wykonania", "Czas wyjścia", "Czas realizacji", "Czas oczekiwania"))

    for p in procesy:
        wynik_plik.write("{:<8} {:<15} {:<15} {:<13} {:<18} {:<18}\n".format(
            p['proces_id'], p['czas_przybycia'], p['czas_wykonania'],
            p['czas_wyjscia'], p['realizacja'], p['oczekiwanie']
        ))

    wynik_plik.write(f"\nŚredni czas oczekiwania: {sredni_czas_oczekiwania:.2f}\n")
```

Dane potrzebne do wykonania zadania zostały umieszczone w pliku o nazwie “dane_procesy.txt” w postaci:

3
0 5
1 8
2 3

gdzie pierwsza cyfra oznacza liczbę procesów, a w każdej kolejnej linijce znajduje się czas przybycia oraz czas wykonania procesu. Algorytm wczytuje te dane do skryptu, a po wykonaniu wyświetla wyniki oraz umieszcza je w osobnym pliku o nazwie “wyniki_sjf.txt”

```
SJF - Wyniki:
Proces   Czas przybycia  Czas wykonania  Czas wyjścia  Czas realizacji    Czas oczekiwania
P3       2               3               5             3                  0
P1       0               5               10            10                 5
P2       1               8               18            17                 9

Średni czas oczekiwania: 4.67
```

```
Proces   Czas przybycia  Czas wykonania  Czas wyjścia  Czas realizacji    Czas oczekiwania
P3       2               3               5             3                  0
P1       0               5               10            10                 5
P2       1               8               18            17                 9

Średni czas oczekiwania: 4.67

Process finished with exit code 0
```

Wyniki zostały sformatowane, aby ułatwić odczyt i zwiększyć przejrzystość.

WNIOSKI

- Algorytm SJF cechuje się niskim średnim czasem oczekiwania
- Przyjmuje procesy bazując na czasie wykonania oraz czasie przybycia
- Większe procesy są umieszczane na końcu, co w przypadku dużej ilości ciągle przybywających, krótszych procesów może doprowadzić do zaniedbania ich

PODSUMOWANIE DLA ALGORYTMÓW PLANOWANIA CZASU PROCESORA

Patrząc na oba algorytmy można stwierdzić, że w większości przypadków lepiej będzie się sprawdzał algorytm SJF, przez jego mniejszy średni czas oczekiwania. Wyjątkami mogą być sytuacje, gdzie przy pojawianiu się dużej ilości mniejszych procesów, procesy większe, które znajdowały się na początku będą spychane na koniec, co może doprowadzić do ich 'zagłodzenia'. Dla wszystkich przykładów poniżej mniejszy średni czas oczekiwania miał SJF.

Przykłady działania obu algorytmów dla różnych zestawów procesów:

Liczba procesów: 4 Dane: 0 6, 0 2, 0 8, 0 3

FCFS - Wyniki:

Proces	Czas przybycia	Czas wykonania	Czas wyjścia	Czas realizacji	Czas oczekiwania
P1	0	6	6	6	0
P2	0	2	8	8	6
P3	0	8	16	16	8
P4	0	3	19	19	16

Średni czas oczekiwania: 7.50

SJF - Wyniki:

Proces	Czas przybycia	Czas wykonania	Czas wyjścia	Czas realizacji	Czas oczekiwania
P2	0	2	2	2	0
P4	0	3	5	5	2
P1	0	6	11	11	5
P3	0	8	19	19	11

Średni czas oczekiwania: 4.50

Liczba procesów: 5 Dane: 0 15, 2 4, 4 3, 6 2, 8 1

FCFS - Wyniki:

Proces	Czas przybycia	Czas wykonania	Czas wyjścia	Czas realizacji	Czas oczekiwania
P1	0	15	15	15	0
P2	2	4	19	17	13
P3	4	3	22	18	15
P4	6	2	24	18	16
P5	8	1	25	17	16

Średni czas oczekiwania: 12.00

SJF - Wyniki:

Proces	Czas przybycia	Czas wykonania	Czas wyjścia	Czas realizacji	Czas oczekiwania
P5	8	1	9	1	0
P4	6	2	11	5	3
P3	4	3	14	10	7
P2	2	4	18	16	12
P1	0	15	33	33	18

Średni czas oczekiwania: 8.00

Liczba procesów: 6 Dane:0 5, 1 9, 2 3, 4 7, 5 2, 6 1

FCFS - Wyniki:

Proces	Czas przybycia	Czas wykonania	Czas wyjścia	Czas realizacji	Czas oczekiwania
P1	0	5	5	5	0
P2	1	9	14	13	4
P3	2	3	17	15	12
P4	4	7	24	20	13
P5	5	2	26	21	19
P6	6	1	27	21	20

Średni czas oczekiwania: 11.33

SJF - Wyniki:

Proces	Czas przybycia	Czas wykonania	Czas wyjścia	Czas realizacji	Czas oczekiwania
P6	6	1	7	1	0
P5	5	2	9	4	2
P3	2	3	12	10	7
P1	0	5	17	17	12
P4	4	7	24	20	13
P2	1	9	33	32	23

Średni czas oczekiwania: 9.50

ALGORYTMY ZASTĘPOWANIA STRON

-- FIFO--

```python
from collections import deque

#Tworzymy zmienne, gdzie mamy ilosc zaladowanych teraz stron, kolejkę i błędy
def bledy_strony_fifo(strony, ilosc, pojemnosc):  1 usage
    pamiec = set()
    kolejka = deque()
    bledy = 0

#Sprawdzamy czy strona znajduje się w pamięci
#Szukamy tej, która jest najstarsza i ją usuwamy, a następnie dodajemy nową do pamięci i kolejki
    for strona in strony:
        if strona not in pamiec:
            if len(pamiec) >= pojemnosc:
                usunieta = kolejka.popleft()
                pamiec.remove(usunieta)

            pamiec.add(strona)
            kolejka.append(strona)
            bledy += 1

    return bledy

#Wczytujemy dane z pliku wewnątrz kodu
with open("dane.txt", "r", encoding="utf-8") as plik:
    linie = [linia.strip() for linia in plik if linia.strip() != '']

```

```python
#Otwieramy plik do zapisu i zapisujemy w nim wyniki
with open("wyniki_fifo.txt", "w", encoding="utf-8") as wynik_plik:
    for i in range(0, len(linie), 2):
        strony = list(map(int, linie[i].split()))
        pojemnosc = int(linie[i + 1])
        ilosc = len(strony)

        wynik = bledy_strony_fifo(strony, ilosc, pojemnosc)

        wynik_plik.write(f"\nFIFO - Zestaw {i // 2 + 1}:\n")
        wynik_plik.write(f"Strony:     {strony}\n")
        wynik_plik.write(f"Pojemność:  {pojemnosc}\n")
        wynik_plik.write(f"Błędy:      {wynik}\n")
```

Dane potrzebne do wykonania zadania zostały umieszczone w pliku o nazwie “dane.txt” w postaci:

1 2 3 4 1 2 5 1 2 3 4 5
3

gdzie na początku widzimy schemat stron pamięci, a liczba pod spodem oznacza pojemność ramek pamięci. Dane są pobierane z pliku i analizowane przez algorytm, a po wykonaniu wyniki zostają umieszczone w pliku “wyniki_fifo.txt”.

```
1   FIFO - Zestaw 1:
2   Strony:     [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]
3   Pojemność:  3
4   Błędy:      9
```

WNIOSKI

- Algorytm FIFO usuwa najstarszą stronę, nie patrząc na stan jej użycia, zostaje ona umieszczana na początku kolejki, a następnie wyrzucana, gdy kolejna strona chce zająć jej miejsce
    - Najnowsza strona jest dodawana na koniec kolejki
- Algorytm jest prosty i intuicyjny, jednak nie uwzględnia użyteczności poszczególnych stron

-- LRU--

```python
#Tworzymy potrzebne zmienne
def bledy_strony_lru(strony, ilosc, pojemnosc):  1 usage
    pamiec = set()
    indeksy = {}
    bledy = 0

#Przechodzimy przez każdą stronę
    for i in range(ilosc):
        strona = strony[i]

#Sprawdzamy czy strona znajduje się w pamięci
#Szukamy tej, która najdłużej nie była używana i ją usuwamy, a następnie aktualizujemy indeks
        if strona not in pamiec:
            if len(pamiec) >= pojemnosc:
                lru = min(pamiec, key=lambda s: indeksy[s])
                pamiec.remove(lru)

            pamiec.add(strona)
            bledy += 1

        indeksy[strona] = i

    return bledy

#Wczytujemy dane z pliku wewnątrz kodu
with open("dane.txt", "r") as plik:
    linie = [linia.strip() for linia in plik if linia.strip() != '']

```

```python
#Otwieramy plik do zapisu i zapisujemy w nim wyniki
with open("wyniki_lru.txt", "w", encoding="utf-8") as wynik_plik:
    for i in range(0, len(linie), 2):
        strony = list(map(int, linie[i].split()))
        pojemnosc = int(linie[i + 1])
        ilosc = len(strony)
        wynik = bledy_strony_lru(strony, ilosc, pojemnosc)


        wynik_plik.write(f"\nLRU - Zestaw {i // 2 + 1}:\n")
        wynik_plik.write(f"Strony:     {strony}\n")
        wynik_plik.write(f"Pojemność:  {pojemnosc}\n")
        wynik_plik.write(f"Błędy:      {wynik}\n")
```

Dane potrzebne do wykonania zadania zostały umieszczone w pliku o nazwie “dane.txt” w postaci:

1 2 3 4 1 2 5 1 2 3 4 5
3

gdzie na początku widzimy schemat stron pamięci, a liczba pod spodem oznacza pojemność ramek pamięci. Dane są pobierane z pliku i analizowane przez algorytm, a po wykonaniu wyniki zostają umieszczone w pliku “wyniki_lru.txt”.

```
1   LRU - Zestaw 1:
2   Strony:     [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]
3   Pojemność:  3
4   Błędy:      10
```

WNIOSKI

- Algorytm LRU bierze pod uwagę użyteczność strony, dzięki temu unikamy usunięcia starej strony, która nadal jest używana
- Do stron przypisywane są indeksy, ułatwiające identyfikacje
- Jest skuteczniejszy niż FIFO, ponieważ porządkuje strony według użycia, a nie czasu, co pozwala na wyrzucenie dawno nieużywanych stron

PODSUMOWANIE DLA ALGORYTMÓW ZASTĘPOWANIA STRON

Patrząc na oba algorytmy można stwierdzić, że w większości przypadków lepiej będzie się sprawdzał algorytm LRU, przez to, że unikamy dzięki niemu usuwania stron nadal użytkowanych. W większości przypadków ma on również mniejszą ilość błędów, chociaż nie zawsze się to sprawdza (tak jak w przykładzie 1). Algorytm FIFO jest prostszy i łatwiejszy do zaimplementowania, jednak mniej skuteczny.

Przykłady działania obu algorytmów dla różnych parametrów stron

Dane użyte dla obu algorytmów:

```
1 2 3 4 1 2 5 1 2 3 4 5
3

7 0 1 2 0 3 0 4 2 3 0 3 2
4

1 2 1 3 1 2 4 5 2 1 2 3 4 5
3
```

Na górze - schemat stron pamięci, na dole - pojemność ramek pamięci

Wyniki dla FIFO:

```
FIFO - Zestaw 1:
Strony:     [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]
Pojemność:  3
Błędy:      9

FIFO - Zestaw 2:
Strony:     [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
Pojemność:  4
Błędy:      7

FIFO - Zestaw 3:
Strony:     [1, 2, 1, 3, 1, 2, 4, 5, 2, 1, 2, 3, 4, 5]
Pojemność:  3
Błędy:      10
```

Wyniki dla LRU

```
LRU - Zestaw 1:
Strony:     [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]
Pojemność:  3
Błędy:      10

LRU - Zestaw 2:
Strony:     [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
Pojemność:  4
Błędy:      6

LRU - Zestaw 3:
Strony:     [1, 2, 1, 3, 1, 2, 4, 5, 2, 1, 2, 3, 4, 5]
Pojemność:  3
Błędy:      9
```