30.01.2021 Politechnika Wrocławska

Sprawozdanie

do zadania ***Programy symulujące***
z przedmiotu

Systemy operacyjne

Politechnika Wrocławska

Yevchuk Mykhailo,
Cyberbezpieczeństwo, Wydział Elektroniki, PWr

W ramach tego zadania zrealizowałem dwa algorytmy przydziału czasu procesora oraz porównałem ich.

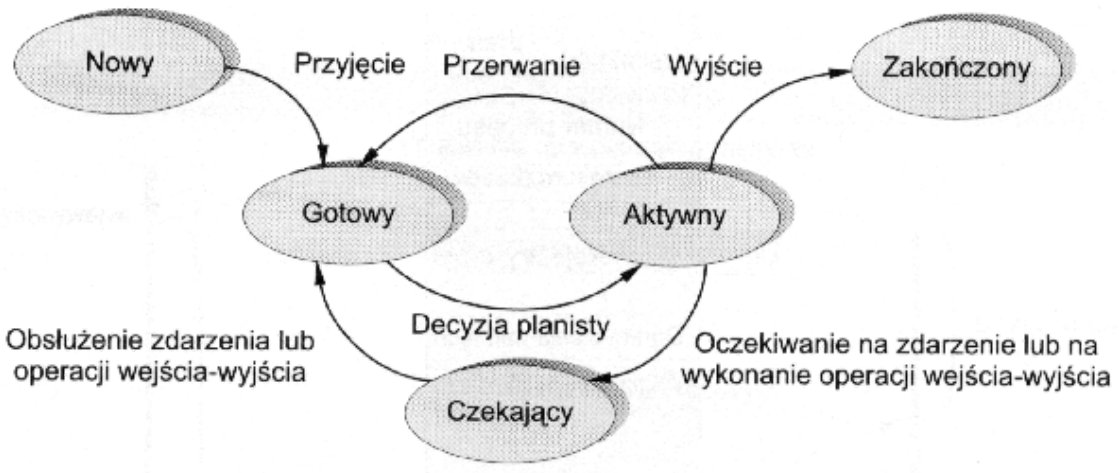
Algorytmy przydziału czasu procesora

Proces – egzemplarz wykonywanego programu. Aplikacja może składać się z większej liczby procesów. Wykonanie instrukcji jednego procesu musi być sekwencyjne, inaczej mówiąc instrukcje są wykonywane w określonej kolejności. Za zarządzanie procesami odpowiada jądro systemu operacyjnego, sposób ich obsługi jest różny dla różnych systemów operacyjnych. Każdy proces otrzymuje od systemu operacyjnego odrębne zasoby, w tym odrębną przestrzeń adresową, listę otwartych plików, urządzeń itp.

Stan procesu - w każdej chwili proces jest w jakimś stanie. Ten stan zmienia się w miarę postępu wykonania procesu.Proces może znajdować się w takich stanach:

- nowy - proces właśnie utworzono.
- aktywny - proces jest właśnie wykonywany przez processor.
- czekający - Proces czeka na zajście jakiegoś zdarzenia (np. wykonanie operacji wejścia-wyjścia).
- gotowy - proces czeka na przydzielenie mu procesora.
- zakończony - proces zakończył działanie.

Diagram stanów procesu



Planowanie procesów - polega na wskazywaniu procesu, któremu ma być w danej chwili przydzielony procesor. W szczególności oznacza to decydowanie, kiedy i który proces ma przejść ze stanu gotowy do stanu aktywny. W systemie w każdej chwili może być aktywnych co najwyżej tyle procesów ile jest procesorów. (Przy implementacji algorytmów planowania zakładałem, dla uproszczenia, że system ma tylko jeden procesor oraz że procesy nie przechodzą w stan oczekiwania na zasoby) Wszystkie pozostałe procesy muszą czekać na przydział procesora.

Planowanie - celem planowania jest maksymalizacja czasu wykorzystania procesora przy wieloprogramowości.Strategie planowania można oceniać z różnych punktów widzenia. Przyjęcie konkretnego kryterium może całkowicie zmienić naszą ocenę danej strategii.

Kryteria oceny algorytmów planowania:

- Wykorzystanie procesora - wykorzystanie procesora powinno być jak największe.
- Przepustowość - liczba procesów kończących się w jednostce czasu.
- Czas cyklu przetwarzania - czas potrzebny na wykonanie procesu (Od momentu pojawienia się w systemie do zakończenia jego wykonania).
- Czas oczekiwania - czas, który proces spędza czekając w kolejce procesów gotowych.
- Czas odpowiedzi - czas pomiędzy wysłaniem żądania a pojawieniem się pierwszej odpowiedzi.

Wymagania stawiane algorytmom planowania:

- Wszystkie systemy:
    - Sprawiedliwość – każdy proces dostaje odpowiedni dla siebie czas procesora.
        - Czas oczekiwania na przydzielenie CPU jest skończony.
    - Wymuszanie strategii – sprawdzanie czy założona strategia szeregowania jest wykonywana.
    - Równowaga – system i jego komponenty są równomiernie obciążone.
- Systemy wsadowe:
    - Jak największa przepustowość.
    - Krótki czas cyklu przetwarzania.
    - Pełne wykorzystanie procesora.
- Systemy Interaktywne:
    - Jak najkrótszy czas odpowiedzi.
        - Procesy pierwszoplanowe mają pierwszeństwo nad drugoplanowymi.
- Systemy czasu rzeczywistego:
    - Dotrzymywanie terminów.

FCFS

FCFS (*First-Come, First-Served*) – czyli "pierwszy przyszedł pierwszy obsłużony", to strategia szeregowania bez wywłaszczania. Procesy są wykonywane od początku do końca w takiej kolejności, w jakiej pojawiły się w kolejce gotowych. Algorytm ten dokonuje najsprawiedliwszego przydziału czasu (każdemu według potrzeb), jednak powoduje bardzo słabą interakcyjność systemu – pojedynczy długi proces całkowicie blokuje system na czas swojego wykonania, gdyż nie ma priorytetów zgodnie z którymi mógłby zostać wywłaszczony.

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

Gantt Chart :

P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :

(0+24+27)/3 = 17

Implementacja.

Implementację danego algorytmu dokonałem w języku Python.Główną ideą przedstawienia każdego procesu było stworzenie słownika, w którym klucze to są parametry opisujące proces, a wartości - dane odpowiadające parametrom. Procesy będą przechowywane w liście, gdzie każdy element listy będzie słownikiem(tzn. osobnym procesem).

Funkcja createRandomlyProcessesList(size:int):

Ponieważ będziemy testować algorytm na odnośnie dużej ilości procesów, stworzyłem funkcję dla wypełnienia listy procesami z losowymi wartościami.Funkcja przyjmuje liczbę typu *int* o nazwie size, która reprezentuję żądaną długość listy procesów, czyli tak naprawdę ta liczba wskazuję na ilość procesów do stworzenia.Funkcja zwraca listę stworzonych procesów.Procesy będziemy charakteryzować sześcioma wartościami, ale nadać procesowi przy stworzeniu powinniśmy tylko 3 wartości:

- "Process ID" - unikalny identyfikator tego procesu (pierwszy proces otrzymuje identyfikator równy 1, identyfikatory następnych procesów inkrementujemy).
- "Arrival Time" - czas przybycia tego procesu, czyli moment, kiedy proces będzie gotowy do wykonania.Ta wartość jest ustawiana losowo, od 0 do maksymalnie możliwej długości wykonywania wszystkich procesów.
- "Burst Time" - czas potrzebny na wykonanie danego procesu.Wartość ustawiamy losowo, od 0 do 20[ms] (jednostki są warunkowe, ale możemy przyjąć że obliczamy czas systemu w milisekundach).

Pozostałe wartości procesów będą kalkulowane w trakcie ich wykonywania, na podstawie przydzielonych wartości losowych.

```python
def createRandomlyProcessesList(size:int):
    """ Returns a list of dictionaries
        where every dictionary is a single process with all necessary information.
        All values in the dictionary are set randomly """

    #maxBurstTime variable will be used to set an arrival time of the process
    #this value is not specified so you can change it, but I found the value of (20 * size)
    the most optimal

    maxBurstTime = 20 * size
    processesList = []
    processTemplate = {"Process ID": 0,
                       "Arrival Time": 0,
                       "Burst Time": 0,
                       "Completion Time": 0,
                       "Turnaround Time": 0,
                       "Waiting Time": 0}

    #here we fill up our list with randomly created processes and append it as a dictionary
    #I used copy() method because it returns another object with different id, and this is
    what I needed

    for i in range(size):
        processesList.append(processTemplate.copy())
        processesList[i]["Process ID"] = i + 1
        processesList[i]["Arrival Time"] = random.randint(0, maxBurstTime)
        processesList[i]["Burst Time"] =  random.randint(1, 20)

    return processesList
```

Funkcja createManuallyProcessesList():
Funkcja, zarówno jak i funkcja opisana wyżej, wykonuje wypełnienie listy procesów procesami, tylko że robi to nie w sposób losowy, a w sposób komunikacji z użytkownikiem. Funkcja zwraca listę wypełnioną procesami. Pytamy użytkownika o ilości procesów, po czym tworzymy listę procesów o zadanej długości.Potem w pętli pytamy o czas przybycia procesu oraz czas jego wykonywania.Identyfikator ID przydzielamy ze zbioru {1, 2, ... , n}, gdzie n to ilość procesów.Pozostałe wartości,

podobno funkcji opisanej wyżej, będą kalkulowane w trakcie ich wykonywania, na podstawie przydzielonych wartości od użytkownika.

```python
def createManuallyProcessesList():
    """ Returns the list of dictionaries
        where every dictionary is a single process with all necessary information.
        Values in the dictionary are set by user's input """

    numberOfProcesses = int(input("How many processes you want to create: "))
    processTemplate = {"Process ID": 0,
                       "Arrival Time": 0,
                       "Burst Time": 0,
                       "Completion Time": 0,
                       "Turnaround Time": 0,
                       "Waiting Time": 0}

    processesList = [processTemplate.copy() for i in range(numberOfProcesses)]

    #I ask only for two parameters because others will be calculated based on these two
    for process in range(numberOfProcesses):
        arrivalTime = int(input("P{proc} Enter arrival time:".format(proc=process + 1)))
        burstTime = int(input("P{proc} Enter burst time:".format(proc=process + 1)))

        processesList[process]["Arrival Time"] = arrivalTime
        processesList[process]["Burst Time"] = burstTime
        processesList[process]["Process ID"] = process + 1

    return processesList
```

Funkcja getAverageValuesOfProcesses(processesList:list):
Wywołanie tej funkcji zwróci nam średnie wartości czasu oczekiwania każdym procesem na wykonywanie (będąc w kolejce gotowych do wykonywania procesów) oraz czasu cyklu przetwarzania.Tym samym dana funkcja imituję procesor, który wykonuje procesy z listy procesów.Funkcja przyjmuje wartość processesList typu *list*. Ponieważ nasz algorytm zakłada że proces jest wykonywany od początku do końca, logika funkcji jest prosta: najpierw sortujemy listę procesów według ich czasu przybycia. Zatem przerabiamy każdy proces w pętli, razem z tym licząc pozostałe parametry każdego procesu i dodajemy te wartości do zmiennych averageWaitingTime oraz averageTurnaroundTime. Po zakończeniu procesowania procesów zwracamy wartości średnie jako krotkę.

```python
def getAverageValuesOfProcesses(processesList:list):
    """ Returns a tuple with average waiting time and average turnaround time of taken processes """

    currentTimeCPU = 0
    averageWaitingTime = 0
    averageTurnaroundTime = 0

    #I used lambda function to sort dictionaries by arrival time for easier processing
    processesList = sorted(processesList, key = lambda i: i["Arrival Time"])

    for process in range(len(processesList)):
        if processesList[process]["Arrival Time"] > currentTimeCPU:
            currentTimeCPU = (processesList[process]["Arrival Time"] + processesList[process]["Burst Time"])
        else:
            currentTimeCPU += processesList[process]["Burst Time"]

        processesList[process]["Completion Time"] = currentTimeCPU

        processesList[process]["Turnaround Time"] = (
            processesList[process]["Completion Time"] - processesList[process]["Arrival Time"])

        processesList[process]["Waiting Time"] = (
            processesList[process]["Turnaround Time"] - processesList[process]["Burst Time"])

        averageWaitingTime += processesList[process]["Waiting Time"]
        averageTurnaroundTime += processesList[process]["Turnaround Time"]

    try:
        averageWaitingTime /= len(processesList)
        averageTurnaroundTime /= len(processesList)
    except ZeroDivisionError:
        print("There are no processes to process!")

    return (averageWaitingTime, averageTurnaroundTime)
```

Funkcje do zapisu danych do pliku oraz odczu danych z pliku

Także realizowałem osobne proste funkcje dla zapisywania listy procesów do pliku CSV oraz odczytu z takiego pliku.

```python
def writeProcessesDataToCSVfile(processesList:list, fileName:str="processesDataFCFS.csv"):
    """ Writes down all the information about single process to a CSV file given as argument,
    if file wasn't given as argument -
    creates new CSV file named 'processesDataFCFS.scv' and writes down information to it """
    CSVcolumns = ["Process ID", "Arrival Time", "Burst Time", "Completion Time", "Turnaround
    Time", "Waiting Time"]
    try:
        with open(fileName, "w+") as f:
            writer = csv.DictWriter(f, fieldnames=CSVcolumns)
            writer.writeheader()
            for process in processesList:
                writer.writerow(process)
        print("Successfully written to", fileName)
    except IOError:
        print("Input/Output error!")
```

```python
def readProcessesDataFromCSVfile(fileName:str):
    """ Reads information about processes from CSV file. Returns a list where every single
    process is a dictionary. """
    try:
        with open(fileName, "r") as f:
            processesList = [{key: int(value) if value.isdigit() else float(value) for key,
            value in row.items()}
            for row in csv.DictReader(f, skipinitialspace=True)]

        print("Successfully read from", fileName)
        return processesList
    except IOError:
        print("Input/Output error!")
```

Wyniki:
Po uruchomieniu programu w pętli 100 razy z listami składającymi się ze 100 procesów z losowo ustawionymi danymi otrzymujemy następujące wyniki:

```
Successfully written to processesDataFCFS.csv
from 100 different lists of 100 operations in every we get:
Average Waiting Time = 7.257100000000001
Average Turnaround Time = 17.764999999999997
```

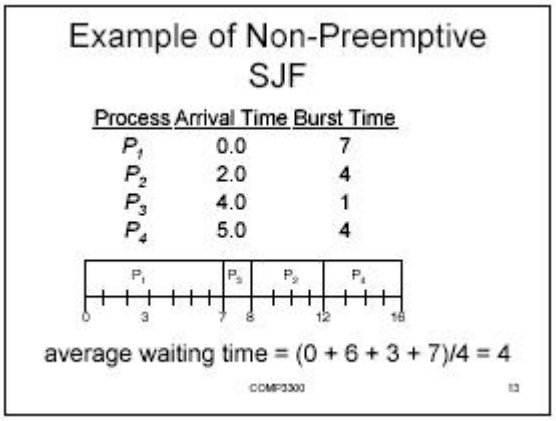
Te wyniki będziemy mogli porównać z wynikami otrzymanymi z działania następnego algorytmu przydziału czasu procesora SJF.

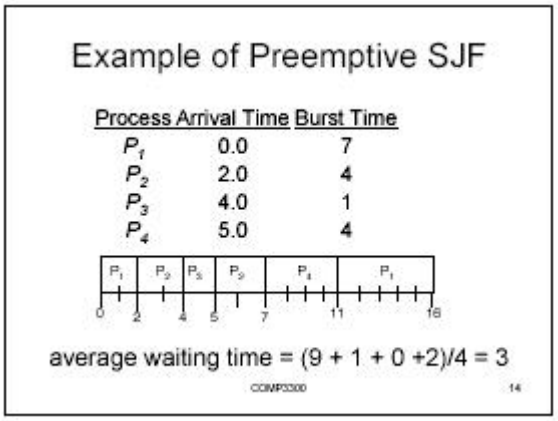
SJF

SJF (*Shortest-Job-First*) – czyli "najpierw najkrótsze zadanie", jest algorytmem optymalnym ze względu na najkrótszy średni czas oczekiwania. W wersji z wywłaszczaniem (istnieje również wersja bez wywłaszczania), stosowana jest metoda: najpierw najkrótszy czas pracy pozostałej do wykonania. Problemem tego algorytmu jest głodzenie długich procesów – może się zdarzyć, że cały czas będą nadchodzić krótsze procesy, a wtedy proces dłuższy nigdy nie zostanie wykonany.

Planowanie priorytetowe:

- SJF jest przykładem algorytmu priorytetowego:
    - Priorytet jest odwrotnie proporcjonalny do czasu trwania kolejnej fazy CPU.
- Głodzenie:
    - Procesy o niskim priorytecie mogą nigdy nie zostać dopuszczone do procesora.
    - Problem można rozwiązać poprzez postarzanie – stopniowe podwyższanie priorytetu procesów długo oczekujących.





Implementacja.
Implementację danego algorytmu też dokonałem w języku Python.Główna idea przedstawienia każdego procesu pozostała taka sama: stworzenie słownika, w którym klucze to są parametry opisujące proces, a wartości - dane odpowiadające parametrom. Procesy będą przechowywane w liście, gdzie każdy element listy będzie słownikiem(tzn. osobnym procesem).

Funkcje createRandomlyProcessesList(size:int) oraz createManuallyProcessesList() pozostałe dosłownie takie same jak w implementacji algorytmu FCFS, dodałem tylko jedną wartość do słownika reprezentującego proces, ponieważ zrealizowałem wersje algorytmu z wywłaszczeniem i będę jej potrzebował dla kalkulacji ile czasu proces już był przetwarzany a ile mu jeszcze zostało.

```
processTemplate = {"Process ID": 0,
                   "Arrival Time": 0,
                   "Burst Time": 0,
                   "Completion Time": 0,
                   "Turnaround Time": 0,
                   "Waiting Time": 0,
                   "Time Of Process Execution": 0}
```

Ta jedna dodatkowa wartość to "Time Of Process Execution".

Funkcja getAverageValuesOfProcesses(processesList:list):
Główna idea tej funkcji została taka sama jak w implementacji algorytmu FCFS, czyli na wejściu ona otrzymuje listę procesów a na wyjściu zwraca wartości średnie jako krotkę (średni czas oczekiwania oraz średni czas cyklu przetwarzania dla procesów w liście wejściowej).Ale zmieniłem sposób przetwarzania procesów. Zrealizowałem następny pomysł: odsortowałem wszystkie procesy zgodnie z ich czasem przybycia,

oraz utworzyłem nową listę `arrivedProcessesList = []`, do której będę dodawał wszystkie przebywające procesy.Zatem, po przybyciu nowego procesu będę sortował tą listę zgodnie z czasem wykonania procesów oraz czasem oczekiwania procesu (sortowanie według dwóch kluczy słownika, potrzebuję tego ponieważ w sytuacji kiedy będę miał kilku procesów o najkrótszym czasie przetwarzania muszę wybrać ten który najdłużej czeka) i będę wykonywał najkrótszy ze wszystkich.Jeżeli nowy proces, który przybył i został dołączony do listy procesów przychodzących okaże się najkrótszy, to funkcja zaczyna wykonywać go, przy czym proces, który był wykonywany przed tym zostaje odłożony. Dla zapisu ile jeszcze zostało potrzeba czasu na zakończenie przetwarzania procesu, który został odłożony właśnie potrzebujemy nową wartość `"Time Of Process Execution"`. Kiedy proces zostanie zakończony, jego zaktualizowane dane zapisujemy do listy wszystkich procesów i usuwamy go z listy procesów przebywających.
Utworzyłem zmienną `performedProcesses = 0` która będzie służyła do obliczenia procesów, które już zostały przetworzone.Główną pętlę zrealizowałem jako pętlę *while* w której będziemy przetwarzali procesy.Kiedy proces zostanie zakończony, zaktualizujemy jego dane w liście wszystkich procesów, usuniemy go z listy dla procesów przychodzących oraz zinkrementujemy licznik *performedProcesses*.

```
while performedProcesses != len(processesList):
```

W przypadku jeżeli lista dla procesów przychodzących jest pusta, dodajemy do niej następny proces z listy procesów (ponieważ jest ona posortowana względem czasu przybycia procesu) i sprawdzamy czy następne procesy nie mają tego samego czasu przybycia, jeżeli mają - dodajemy ich również.Po dodaniu procesu (lub kilku) sortujemy listę procesów przychodzących (czyli lista w której wszystkie procesy są gotowe do wykonywania) względem czasu wykonywania oraz czasu oczekiwania (musimy zanegować wartość czasu oczekiwanie powieważ przy negowanie najdłuższa rzeczywista wartość oczekiwania zostanie najmniejszą względem sortowania).

```
#if shortest process that is ready for execution (already stored in arrivedProcessesList)
will finish his execution by arrival of the next process
#we can execut it and delete from arrivedProcessesList, as it is already done

#in other case, if arrival time of next process will be in less time than out currently
shortest process will need for it's execution
#wi will execute only part of it and will stop at the moment when next process will
arrive, then we will add new arrived process to arrivedProcessesList and again will find
the shortest

while performedProcesses != len(processesList):
    if not arrivedProcessesList:
        currentTimeCPU = processesList[arrivedProcessesCounter]["Arrival Time"]

        nextProcessArrivalTime = processesList[arrivedProcessesCounter]["Arrival Time"]
        arrivedProcessesList.append(processesList[arrivedProcessesCounter].copy())
        arrivedProcessesCounter += 1

        #after adding next by arrival time process we should sheck if there are more
        processes with same arrival time and add them
        while (arrivedProcessesCounter < len(processesList)) and ((processesList
        [arrivedProcessesCounter]["Arrival Time"] == nextProcessArrivalTime)):
            arrivedProcessesList.append(processesList[arrivedProcessesCounter].copy())
            arrivedProcessesCounter += 1

    #arrivedProcessesList = sorted(arrivedProcessesList, key=itemgetter('Burst Time',
    'Waiting Time'))
    arrivedProcessesList = sorted(arrivedProcessesList, key=lambda k: (k["Burst Time"], -k
    ["Waiting Time"]))
```

W przypadku, jak nasza lista procesów przychodzących nie jest pusta, powinniśmy wykonywać ten, który stoi na pierwszym miejscu listy, czyli ma indeks 0.
Jeżeli to jest nasz ostatni proces y listy wszystkich procesów lub czas przybycia następnego jest większy niż czas wykonania obecnie najkrótszego to możemy wykonać proces z listy arrivedProcessesLis pod numerem indeksu 0 i po aktualizacji danych usunąć go z listy procesów aktywnych.Również musimy zaktualizować czas oczekiwania wszystkich pozostałych procesów z tej listy, ponieważ były one gotowe do wykonywania.Po usunięciu procesu inkrementujemy licznik przerobionych procesów.

```
for process in range(1, len(arrivedProcessesList)):
    arrivedProcessesList[process]["Waiting Time"] += arrivedProcessesList[0]["Burst Time"]

del arrivedProcessesList[0]
performedProcesses += 1
```

Jeżeli czas przybycia następnego procesu jest mniejszy niż czas, którego najkrótszy proces potrzebuje na wykonanie, to przerabiamy ten proces do momentu przybycia następnego i potem znowu sortujemy listę gotowych do wykonywania procesów żeby znaleźć najkrótszy.

```
    if (arrivedProcessesCounter == len(processesList)) or  (processesList
    [arrivedProcessesCounter]["Arrival Time"] > (arrivedProcessesList[0]["Burst Time"] +
    currentTimeCPU)):

        arrivedProcessesList[0]["Waiting Time"] = (
            currentTimeCPU - arrivedProcessesList[0]["Arrival Time"] -
            arrivedProcessesList[0]["Time Of Process Execution"])
        currentTimeCPU += arrivedProcessesList[0]["Burst Time"]

        arrivedProcessesList[0]["Completion Time"] = currentTimeCPU
        arrivedProcessesList[0]["Turnaround Time"] = (
            arrivedProcessesList[0]["Completion Time"] - arrivedProcessesList[0]["Arrival
            Time"])

        averageWaitingTime += arrivedProcessesList[0]["Waiting Time"]
        averageTurnaroundTime += arrivedProcessesList[0]["Turnaround Time"]

        for process in range(len(processesList)):
            if processesList[process]["Process ID"] == arrivedProcessesList[0]["Process
            ID"]:
                processesList[process] = arrivedProcessesList[0].copy()
                break

        for process in range(1, len(arrivedProcessesList)):
            arrivedProcessesList[process]["Waiting Time"] += arrivedProcessesList[0]
            ["Burst Time"]

        del arrivedProcessesList[0]
        performedProcesses += 1
    else:
        for process in range(1, len(arrivedProcessesList)):
            arrivedProcessesList[process]["Waiting Time"] += (processesList
            [arrivedProcessesCounter]["Arrival Time"] - currentTimeCPU)

        arrivedProcessesList[0]["Time Of Process Execution"] += (processesList
        [arrivedProcessesCounter]["Arrival Time"] - currentTimeCPU)
        arrivedProcessesList[0]["Burst Time"] -= (processesList[arrivedProcessesCounter]
        ["Arrival Time"] - currentTimeCPU)

        currentTimeCPU = processesList[arrivedProcessesCounter]["Arrival Time"]

        arrivedProcessesList.append(processesList[arrivedProcessesCounter])
        arrivedProcessesCounter += 1

averageWaitingTime /= len(processesList)
averageTurnaroundTime /= len(processesList)

return (averageWaitingTime, averageTurnaroundTime)
```

Dla tego algorytmu również realizowałem osobne proste funkcje dla zapisywania listy procesów do pliku CSV oraz odczytu z takiego pliku.Są one takie same jak dla algorytmu FCFS.

Wyniki:

Po uruchomieniu programu w pętli 100 razy z listami składającymi się ze 100 procesów z losowo ustawionymi danymi otrzymujemy następujące wyniki:

```
Successfully written to processesDataSJF.csv
from 100 different lists of 100 operations in every we get:
Average Waiting Time = 4.975399999999999
Average Turnaround Time = 15.566300000000002
```

Porównywanie algorytmów przydziału czasu procesora SJF i FCFS

Kryteria oceny algorytmów:

- Średni czas oczekiwania na przydzielenie procesora.
- Średni czas cyklu przetwarzania.

Widzimy że stosując algorytm SJF czas oczekiwania procesów jest ***mniejszy o 31,44%*** od czasu oczekiwania procesów przy stosowaniu algorytmu FCFS.

W przypadku czasu cyklu przetwarzania to ten czas przy stosowaniu algorytmu SJF jest ***o 12,38% mniejszy*** niż przy stosowaniu algorytmu FCFS.

Wnioskujemy, że ogólnie ***algorytm SJF (szczególnie jego wersja z wywłaszczeniem) jest algorytmem bardziej optymalnym jak ze względu na krótszy średni czas oczekiwania tak i ze względu na krótszy czas cyklu przetwarzania.*** Wynika to z tego że algorytm FCFS dokonuje najsprawiedliwszego przydziału czasu (każdemu według potrzeb), jednak powoduje bardzo słabą interakcyjność systemu – pojedynczy długi proces całkowicie blokuje system na czas swojego wykonania, gdyż nie ma priorytetów zgodnie z którymi mógłby zostać wywłaszczony.