Algorytmy planowania czasu procesora

Ich implementacja i testowanie

Wprowadzenie do systemów komputerowych

Antoni Cichoń (279471) | 07.06.2024

Cyberbezpieczeństwo (I st.) II semestr

Oświadczam, że sprawozdanie zostało przeze mnie wykonane osobiście i samodzielnie

Spis treści

Informacje wstępne

Celem tej symulacji jest poznanie właściwości podstawowych algorytmów planowania czasu procesora.

Do stworzenia symulacji wybrałem język Python. Głównym powodem była obsługa wielu pomocnych bibliotek. Symulacja również nie miała być optymalizowana pod kątem wydajności czy złożoności obliczeniowej, co utwierdziło mnie w przekonaniu że Python będzie odpowiednim wyborem. Do napisania programu symulacyjnego użyłem bibliotek Matplotlib oraz NumPy.

Program symulacyjny najpierw generuje zestaw danych testowych, a następnie przeprowadza testy na każdym z zadanych algorytmów. Wyniki są zapisywane w plikach .csv oraz generowane są histogramy prezentujące uzyskane dane w czytelny sposób.

Generacja danych testowych

Generator danych przyjmuje 3 argumenty:

1. liczbę procesów do wygenerowania;
2. listę zakresów czasów procesów, w kolejności:
    - minimalny czas przybycia,
    - maksymalny czas przybycia,
    - minimalny czas przetwarzania,
    - maksymalny czas przetwarzania;
3. (opcjonalnie) ziarno dla generatora liczb losowych. Przy pominięciu generator używa losowego ziarna.

W swoich testach użyłem poniższych wartości:

1. liczba procesów do wygenerowania: 100;
2. zakresy czasów:
    - przybycia: 1 – 200,
    - przetwarzania: 1 – 200;
3. ziarno: 1234.

Generator najpierw usuwa wszystkie istniejące foldery od „test1” do „testn”, gdzie n jest ostatnim testem w folderze symulacji. Następnie generowane są nowe dane testowe. Użytkownik ma wpływ na sposoby generacji danych poprzez modyfikację słownika „genfunctions”, gdzie kluczem jest typ generacji danych, a wartością funkcja generująca. Użytkownik może usuwać algorytmy generujące oraz dodawać własne. Ja użyłem poniższych algorytmów:

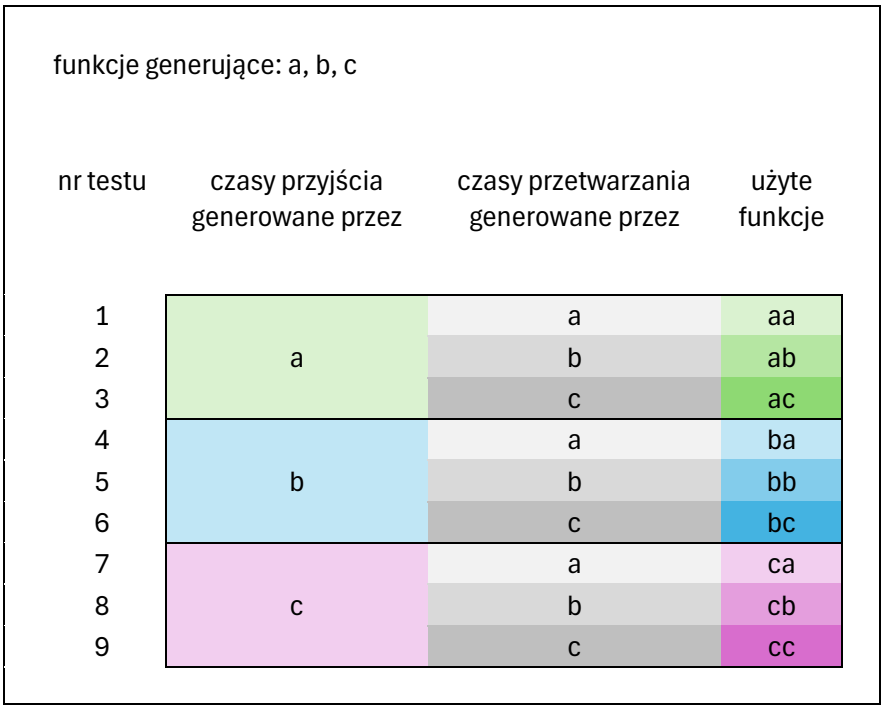
```
genfunctions = {
    "randomized" : random,
    "in normal distribution" : binomial,
    "in exponential distribution" : exponential,
    "in ascending order" : ascending,
    "in descending order" : descending
}
```

inputgen.py, line 16

Każda z podanych funkcji zwraca listę n liczb z podanego zakresu (funkcje „ascending” i „descending” ignorują ten zakres) wygenerowanych w następujący sposób:

1. „random” – liczby generowane całkowicie losowo;
2. „binomial” – liczby generowane wg rozkładu dwumianowego;
3. „exponential” – liczby generowane wg rozkładu wykładniczego;
4. „ascending” – liczby zapętlone w kolejności od najmniejszej do największej;
5. „descending” – liczby zapętlone w kolejności od największej do najmniejszej.

Każdy kolejny test jest wariacją (z powtórzeniami) ze zbioru funkcji generujących. Poniżej przedstawiłem teoretyczny schemat generacji danych z użyciem trzech funkcji generujących: „a”, „b” i „c”:



Z powyższego wynika, że liczbę wygenerowanych testów można opisać wzorem:

$$T = n^k$$

Gdzie:

T – Liczba wygenerowanych testów

n – liczba funkcji generujących

k – liczba użytych funkcji generujących

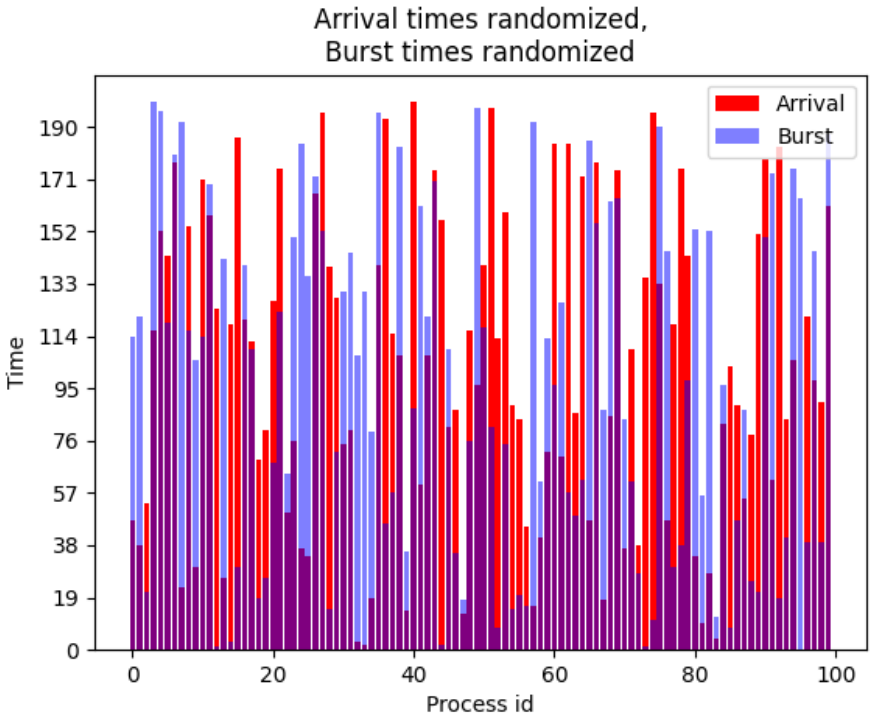
Dla tego generatora wraz z danymi funkcjami generującymi liczba wygenerowanych testów wynosi 5^2, czyli 25.

Po wygenerowaniu zestawu danych testowych generator tworzy w folderze symulacji folder o nazwie „testx”, gdzie x jest numerem algorytmu, za pomocą którego generowane były procesy. Dane testowe zapisywane są do pliku „input.txt”. Pierwsze dwie linijki tego pliku zawierają sposób generacji danych. Każda kolejna linijka zawiera kolejno id procesu, czas przyjścia i czas przetwarzania. Każda z liczb jest oddzielona średnikiem. Dodatkowo dla każdego testu tworzony jest histogram o nazwie „inputvisual.png”, wizualizujący czasy przyjścia i przetwarzania każdego procesu w wygenerowanym teście.

Na koniec generator zwraca listę ścieżek do utworzonych folderów testowych. Program informuje użytkownika o każdym stworzonym folderze. W przypadku niepowodzenia przekazuje, którego testu nie mógł stworzyć oraz pełną ścieżkę do wadliwego folderu i awaryjnie kończy swoje działanie. Kontynuowanie generacji teoretycznie może nastąpić, jednak zdecydowałem się przerwać program ze względu na potencjalne wystąpienie niezgodności między numeracją testów a zadeklarowaną listą funkcji generujących.

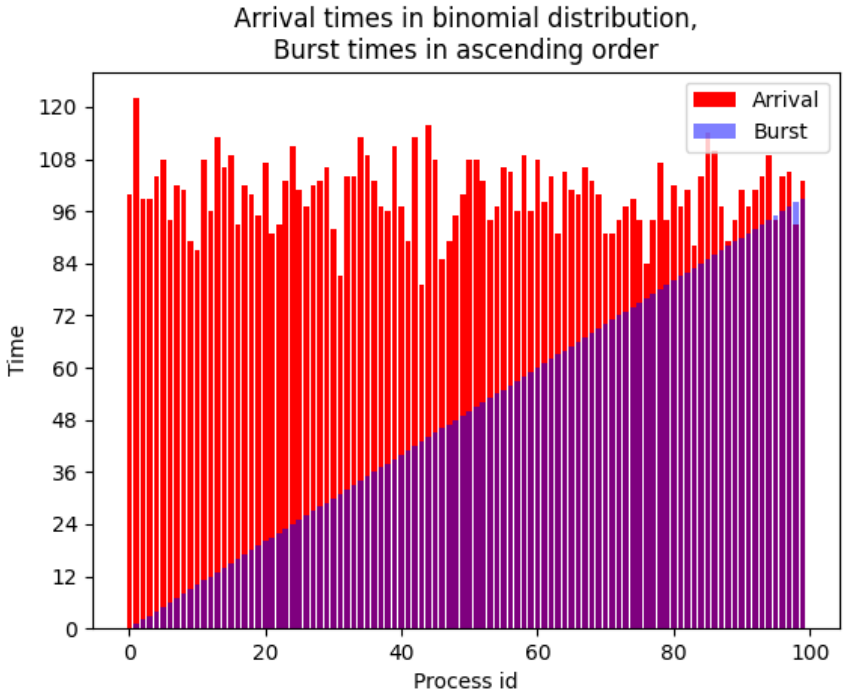
Dla moich danych wejściowych wybrane testy wyglądają następująco:

Test 1:



test1/inputvisual.png

Test 9:



test9/inputvisual.png

Przygotowanie do testu

Aby poprawnie przeprowadzić testy, należy odpowiednio zinterpretować wygenerowane dane. Zdecydowałem się na napisanie klasy „Process” i stworzenie listy obiektów, gdzie obiektem jest pojedynczy proces:

```python
class Process:
    def __init__(self, id, arrivalTime, burstTime):
        self.id : int = id
        self.arrivalTime : int  = arrivalTime
        self.burstTime : int = burstTime
        self.turnaroundTime = None
        self.waitTime = None

    def setTimes(self, time):
        self.turnaroundTime = time - self.arrivalTime
        self.waitTime = self.turnaroundTime - self.burstTime

    def exportCSV(self):
        return f'{self.id};{self.waitTime}'
```

makeQueue.py, line 1

Każdy proces ma 5 atrybutów:

1. „id” – id procesu;
2. „arrivalTime” – czas przybycia;
3. „burstTime” – czas przetwarzania;
4. „turnaroundTime” – czas obrotu;
5. „waitTime” – czas oczekiwania.

Atrybuty „id”, „arrivalTime” i „burstTime” ustawiane są podczas inicjalizacji. Atrybuty „turnaroundTime” oraz „waitTime” obliczane są podczas wykonywania symulacji.

Ponadto zaimplementowałem dwie metody:

1. „setTimes” – na podstawie czasu symulacji oblicza czasy obrotu i czekania danego procesu;
2. „exportCSV” – zwraca tekst zawierający id procesu oraz czas czekania oddzielone średnikiem.

Następnie lista obiektów sortowana jest wg czasu przybycia procesów, co pozwala symulować nadchodzenie kolejnych procesów wraz z upływem czasu symulacji

Przeprowadzanie testu

Po przygotowaniu listy procesów do symulacji program ją rozpoczyna. Każdy test przeprowadzany jest na zadanych algorytmach w słowniku „usealgorithms”. Kluczami są nazwy algorytmów, a wartościami funkcje kolejkujące procesy. Ja zaimplementowałem trzy algorytmy:

main.py, line 17

```
usealgorithms = {
    "First come first serve" : fcfs,
    "Last come first serve" : lcfs,
    "Shortest job first" : sjf
}
```

1. „fcfs” – ustawia procesy wg ich kolejności przybycia (rosnąco);
2. „lcfs” – ustawia procesy wg ich kolejności przybycia (malejąco);
3. „sjf” – ustawia procesy wg ich czasu przetwarzania (rosnąco).

Każdy z tych algorytmów (szczegółowo omówionych poniżej) symuluje czas procesora i na bieżąco aktualizuje kolejne procesy na podstawie czasu symulacji.

Algorytmy planowania czasu procesora

Zadaniem algorytmu planowania czasu procesora jest takie ustawienie procesów w kolejce, aby zminimalizować ich czasy oczekiwania. Od wydajności takiego algorytmu zależy łączny czas wykonywania zadanej kolejki procesów.

Definicje

- Czas obrotu procesu

$$t_O = t_S - t_p$$

  gdzie:

  t_o – czas obrotu
  t_s – czas symulacji
  t_p – czas przybycia

- Czas oczekiwania procesu

$$t_c = t_o - t_w$$

  gdzie:

  t_c – czas oczekiwania
  t_o – czas obrotu
  t_w – czas przetwarzania

- **Algorytm niewywłaszczeniowy** – wykonywany proces zostaje zmieniony tylko po jego zakończeniu.
- **Algorytm wywłaszczeniowy** – wykonywany proces może zostać zmieniony pomimo tego, że nie został jeszcze w pełni wykonany.
- **Efekt głodzenia** – wydłużenie czasu oczekiwania procesu, który przybył wcześniej, wynikające z wykonywania najnowszych procesów jako pierwszych.
- **Efekt konwoju** – wydłużenie czasu oczekiwania wielu procesów o małym czasie przetwarzania, wynikające z wykonywania długiego procesu, który przybył wcześniej.

1. First come first serve

To najprostszy algorytm planowania czasu procesora. Ustawia nadchodzące procesy w kolejce wg ich czasu przybycia – czyli „kto pierwszy ten lepszy” .

```python
def fcfs(original_processes):
    processes = copy(original_processes)
    time = 0
    for process in processes:
        time = max(time, process.arrivalTime)
        time += process.burstTime
        process.setTimes(time)

    return [process.exportCSV() for process in sorted(processes, key=lambda x: x.id)]
```

firstcomefirstserve.py, line 3

Aby zapobiec przypadkowej modyfikacji oryginalnych procesów, należy skopiować oryginalną kolejkę do lokalnej zmiennej. Zwykła operacja przypisania płytkiego („=”) spowodowałaby skopiowanie adresu pamięci listy do zmiennej „processes”, zamiast skopiować listę do nowego miejsca w pamięci. Następnie inicjowany jest czas symulacji. Procesy są po kolei „wykonywane” a czas ich przetwarzania dodawany jest do czasu symulacji. W przypadku gdy żaden proces nie oczekuje na wykonanie, czas symulacji ustawiany jest na wartość czasu przyjścia następnego procesu. Po wykonaniu wszystkich procesów algorytm zwraca listę zawierającą id procesów i ich czas oczekiwania, posortowaną rosnąco wg id procesu. Dane są oddzielone średnikami.

Warto zaznaczyć, że w implementacji tego algorytmu nie jest tworzona nowa kolejka dla przybyłych procesów, ponieważ oryginalna lista procesów już jest posortowana wg czasu przybycia. Przy realnym systemie algorytm nie ma dostępu do takiej listy i może jedynie dopisywać do kolejki nowe procesy. Pozwoliłem sobie odejść od rzeczywistej implementacji FCFS (z użyciem osobnej listy na kolejkę) ze względu na brak wpływu mojego rozwiązania na wynik symulacji, której zadaniem jest jedynie podanie czasu oczekiwania danego procesu.

+ prosty w implementacji

+ wydajny w przypadku kolejki, w której czas wykonywania procesów wzrasta

+ odporny na efekt głodzenia

- niewydajny w przypadku kolejki, w której najdłuższe procesy nadchodzą jako pierwsze

- narażony na efekt konwoju

2. Last come first serve

Algorytm bliźniaczo podobny do FCFS, jednak szeregujący procesy w odwrotnej kolejności – od najnowszych do najstarszych.

```python
def lcfs(original_processes):
    processes = copy(original_processes)
    queue = []
    finished = []
    time = 0

    while queue or processes: # break when queue is empty and there are no more processes to execute
        # put processes which already arrived in queue
        if processes:
            queue.extend(updateQueue(time, processes))
            if not queue: # if queue is empty, set simulation time to arrival of next process and add it to queue
                    time = processes[0].arrivalTime
                    queue.append(processes.pop(0))

        # sort queue from latest to oldest process
        queue.sort(key=lambda x: x.arrivalTime, reverse=True)

        process = queue.pop(0) # pop process from queue and handle it
        time += process.burstTime # update time by burst time of current process ("execute it")
        process.setTimes(time) # set turnaround and waiting time of this process
        finished.append(process) # add this process to finished list

    return [process.exportCSV() for process in sorted(finished, key=lambda x: x.id)]

def updateQueue(time, processes): # returns processes that arrived by now
    arrived = [] # initialize arrived list
    for process in processes: # add processes to list untill process with arrival time greater than current time occurs
        if process.arrivalTime > time: break
        arrived.append(processes.pop(0)) # add process and pop it from process list
    return arrived
```

lascomefirstserve.py, line 3

Tak samo jak w przypadku FCFS tworzona jest kopia oryginalnych procesów. Główna pętla algorytmu wykonywana jest, dopóki nie skończą się procesy zarówno we właściwej kolejce („queue”) jak i oryginalnej liście („processes”). Pierwszym krokiem jest aktualizacja kolejki o procesy które mogły przybyć. W tym celu wywoływana jest funkcja „updateQueue”, która po kolei wyciąga procesy z listy, aż do natrafienia na proces, którego czas nadejścia jest większy niż czas symulacji (proces jeszcze nie nadszedł). Następnie lista nowych procesów jest umieszczana w kolejce, a kolejka sortowana malejąco wg czasu przybycia. W przypadku gdy kolejka jest pusta, ale nie nadeszły jeszcze żadne procesy, czas symulacji ustawiany jest na nadejście kolejnego procesu, a proces ten usuwany jest z listy i wpisywany do kolejki. Po wyczyszczeniu listy procesów kolejka wykonuje się do ostatniego procesu. Zwracane są te same dane, co w przypadku FCFS.

+ stosunkowo prosty w implementacji

+ najwydajniejszy w przypadku, kiedy długie procesy nadchodzą jako pierwsze

- niewydajny w przypadku, kiedy długie procesy znajdują się na końcu kolejki

- zagrożony efektem głodzenia i konwoju

3. Shortest job first

To algorytm nieco lepiej zoptymalizowany od pozostałych. Ustawia procesy w kolejce na podstawie ich czasu wykonania, co skutkuje wykonywaniem najkrótszych procesów na początku, zostawiając te dłuższe na koniec. Implementacja jest niemalże identyczna do LCFS, jednak w momencie sortowania kolejki „queue" wartością sortowaną jest „burstTime", czyli czas przetwarzania procesu. Reszta algorytmu pozostaje bez zmian.

```
queue.sort(key=lambda x: x.burstTime)
```

shortesjobfirst.py, line 18

\+ bardzo wydajny, szczególnie dla krótkich procesów

\- praktycznie niemożliwy do implementacji. W rzeczywistości bardzo rzadko czas przetwarzania jest znany przed wykonaniem procesu

\- dłuższe procesy są zagrożone efektem głodzenia

Reprezentacja wyników

Po przeprowadzeniu testu dla każdego algorytmu wyniki zapisywane są w pliku .csv z nazwą odpowiadającą testowanemu algorytmowi (np. Firstcomefirstserve-result.csv). Pierwsza linijka zawiera nagłówki prezentowanych danych, tj. id procesu i jego czas oczekiwania. Każda kolejna linijka zawiera dane, a kolumny rozdzielone są średnikami. Dodatkowo w folderze głównym generowany jest wykres porównujący wydajność każdego z algorytmów o nazwie „resultgraph.png". Program informuje użytkownika o zapisie wyników. W przypadku niepowodzenia obecny test jest pomijany i następuje przejście do kolejnego.

Po przeprowadzeniu testu zawartość jego folderu testu będzie wyglądać w następujący sposób:

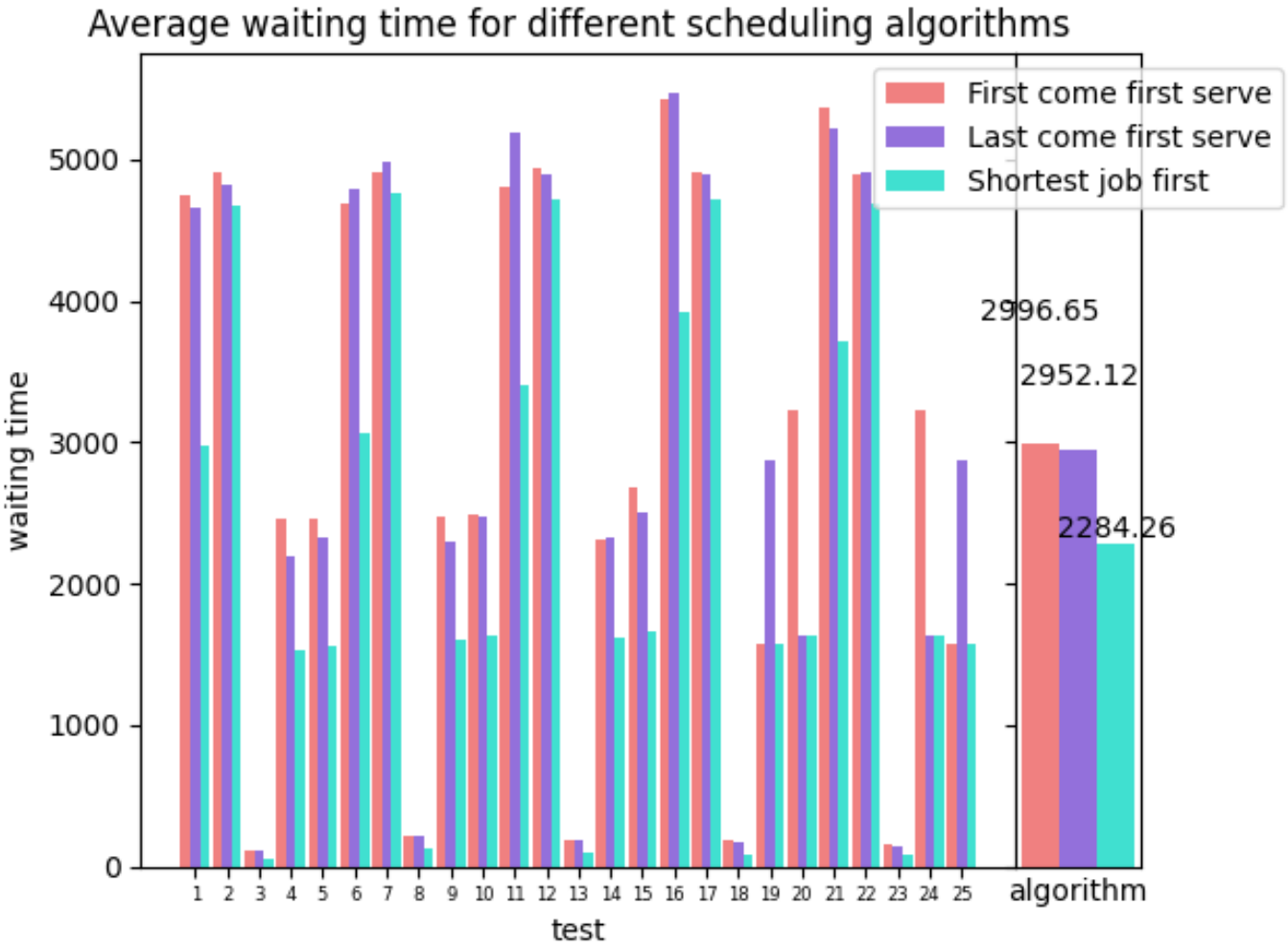
```
C:\Users\Antek\Desktop\pisanie\SK\CPUScheduler\test1>dir /b /oe
Shortestjobfirst-result.csv
Lastcomefirstserve-result.csv
Firstcomefirstserve-result.csv
inputvisual.png
input.txt
```

Z oczywistych powodów ścieżka do folderu będzie się różnić.

Po przeprowadzeniu wszystkich testów (ich lista kontrolowana jest przez generator danych) program oblicza czas wykonania i zwraca go do konsoli.

Interpretacja wyników

Czasy oczekiwania dla poszczególnych algorytmów (przy użyciu ziarna „1234” podczas generacji danych) wygląda następująco:



Na wykresie nr 1 (lewa strona) przedstawione są średnie czasy oczekiwania przez proces w danym teście (oś x). Na wykresie nr 2 (prawa strona) znajduje się uśredniony czas czekania procesu ze wszystkich testów. Pierwsze, co rzuca się w oczy, to oczywista przewaga algorytmu „Shortest job first” nad pozostałymi dwoma. W zależności od testu różnica ta jest większa bądź mniejsza, natomiast SJF ani jeden raz nie wypadł gorzej od FCFS czy LCFS. Biorąc pod uwagę losowość wygenerowanych procesów, można przypuszczać, że SJF najlepiej poradziłby sobie w realnym zastosowaniu.

Różnice między algorytmami FCFS i LCFS zazwyczaj są niewielkie, za wyjątkiem 4 specyficznych testów: 19, 20, 24 i 25. W testach 19 i 25 LCFS wypadł rażąco gorzej od reszty. W testach 20 i 24 ta sama sytuacja dotyczyła FCFS.

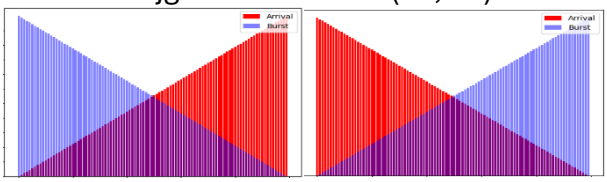
Aby odpowiedzieć na pytanie, dlaczego tak się dzieje, należy przyjrzeć się użytym funkcjom generującym dla tych testów. W tym celu należy spojrzeć na dwie pierwsze linijki pliku „input.txt” w folderze testu.

Dla testu 19: „Arrival times in ascending order, Burst times in ascending order”
Dla testu 20: „Arrival times in ascending order, Burst times in descending order”
Dla testu 24: „Arrival times in descending order, Burst times in ascending order”
Dla testu 25: „Arrival times in descending order, Burst times in descending order”

Biorąc pod uwagę powyższe, można zauważyć, że:

- Algorytm FCFS okazał się skrajnie niewydajny, kiedy procesy przychodzące najwcześniej miały najdłuższy czas przetwarzania.
- Algorytm LCFS okazał się skrajnie niewydajny, kiedy procesy przychodzące najwcześniej miały najkrótszy czas przetwarzania.

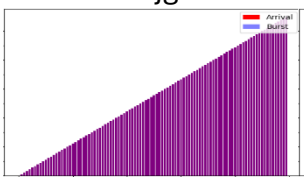
Najgorsze dla FCFS (20, 24)



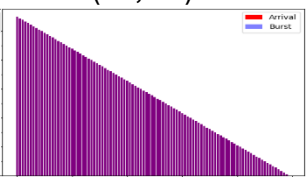
test20/inputvisual.png

test24/inputvisual.png

Najgorsze dla LCFS (19, 25)



test19/inputvisual.png



test25/inputvisual.png

W obu tych przypadkach doszło do efektu konwoju, czyli oczekiwania wielu krótkich procesów na wykonanie kilku długich. FCFS ustawił na początku kolejki długie procesy, ponieważ te nadeszły jako pierwsze, a LCFS zrobił dokładnie to samo, tylko w przypadku, kiedy najdłuższe procesy nadeszły jako ostatnie.

Wnioski

Przed wyciągnięciem wniosków należy wziąć pod uwagę fakt, że powyższe testy są jedynie syntetycznymi scenariuszami i różnią się od realnych sytuacji.

Z przeprowadzonych testów wynika, że algorytm „Shortest job first” zdecydowanie wydajniej tworzył kolejkę procesów. Algorytmy „First come first serve” i „Last come first serve” wypadły podobnie, z minimalną przewagą LCFS. Zgubne dla nich okazały się testy 19, 20, 24 i 25, w których uwydatniły się największe słabości kolejkowania na podstawie czasu przyjścia procesu. Jak widać SJF zdecydowanie lepiej poradził sobie od reszty algorytmów w tych przypadkach (nie był wybitnie wydajny w tych przypadkach, ale również nie był najgorszy w żadnym z nich). Stało się tak, ponieważ niezależnie od czasu nadejścia wykonywał on najkrótsze procesy jako pierwsze, co zapobiegło ich zablokowaniu przez dłuższy proces. Należy jednak pamiętać, że nie zaimplementowałem mechanizmu priorytetyzowania procesów, co mogłoby znacząco zmienić wyniki powyższych testów. Wybranie najlepszego algorytmu wymagałoby pochylenia się również nad algorytmami wywłaszczeniowymi (np. „Round-robin” albo „Shortest remaining time first”), których w tym projekcie nie uwzględniłem.

Podsumowanie

Dobór właściwego algorytmu planowania czasu procesora jest niełatwym zadaniem. Pod uwagę trzeba wziąć zarówno wydajność algorytmów, jak i ich koszt zasobów (aby nie doprowadzić do sytuacji, gdzie algorytm znacząco wydłuża czas wykonania kolejki z powodu prowadzenia skomplikowanych obliczeń). Obiecujący wydawał się algorytm „Shortest job first”, jednak z powodu niewiedzy na temat czasu przetwarzania dla nadchodzących procesów, jest on często niemożliwy do zaimplementowania w realnych systemach.

Użycie języka Python do stworzenia symulacji było odpowiednią decyzją. Dzięki jego prostocie nie musiałem rozwiązywać trywialnych problemów i mogłem skupić się na implementacji algorytmów i generacji danych. Niezwykle przydatna okazała się biblioteka Matplotlib, która umożliwiła czytelną prezentację danych zarówno wejściowych, jak i wyjściowych.