Sprawozdanie

Przeprowadzenie symulacji działania planowania czasu procesora oraz zastępowania stron

Spis treści

[Wstęp 2](#_Toc200210751)

[Symulacja planowania czasu procesora 3](#_Toc200210752)

[Symulacja zastępowania stron 12](#_Toc200210753)

# Wstęp

1. Opis modułów

Program zostały stworzony w oparciu o 4 moduły:

* 1. main.py – moduł główny przeprowadzający symulację, testy
  2. cpuschedule.py – zawiera zaimplementowane algorytmy kolejkowania procesów oraz generator procesów
  3. pagereplacement.py – zawiera zaimplementowane algorytmy zastępowania stron oraz generator stron
  4. graphs.py – odpowiada za rysowanie wykresów dla każdego testu

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

1. Zastosowane algorytmy planowania czasu procesora
   1. Shortest Job First(SJF)
   2. Round Robin(RR)
2. Zastosowane algorytmy zastępowania stron
   1. First In First Out(FIFO)
   2. Least Frequently Used(LFU)
   3. Least Recently Used(LRU)

# Symulacja planowania czasu procesora

1. Opis działania algorytmów SJF i RR

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

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

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

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

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

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

1. Opis procedury testowania

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

Obraz zawierający zrzut ekranu, tekst, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

Każdy test zostaje uruchomiony funkcją *run\_cpu\_tests(),* która przyjmuje ustawienia odnośnie poszczególnego testu. Zostają w niej wygenerowane procesy, które następnie zostają przekazane do funkcji *simulation\_cpu\_schedulding(),* która uruchamia symulację dla zaimplementowanych algorytmów (Rysunek 4.).

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, zrzut ekranu, Czcionka, wyświetlacz

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, zrzut ekranu, wyświetlacz, oprogramowanie

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

1. Przeprowadzone eksperymenty
   1. 25,75,125 procesów, czasy nadejścia (0,100), czas wykonania 5, kwant czasu = 3

Rysunek

* 1. 25,75,125 procesów, czasy nadejścia 0,25,50, czas wykonania = 10, kwant czasu = 5

Rysunek

* 1. 25,75,125 procesów, czas nadejścia 0, czasy wykonania (100,1000), kwant czasu = 300

Rysunek

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

Rysunek

Rysunek

1. Wnioski

Na podstawie wykresów możemy stwierdzić, że istnieje ogromna przepaść wydajnościowa pomiędzy algorytmami, która zwiększa się wraz z ilością stron. Duże czasy oczekiwania i realizacji dla Round Robin wynikają z dużej ilości przełączeń kontekstu, które algorytm musi przeprowadzać po minięciu danego kwantu czasu.

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

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

# Symulacja zastępowania stron

1. Opis algorytmów FIFO, LRU, LFU

**FIFO** – zastępuje „najstarszą” stronę w pamięci.

**LRU** – Zastępuje najdawniej użytą stronę.

**LFU –** zastępuje najrzadziej używaną stronę.

2. Opis procedury testowania

Strony są generowane za pomocą funkcji *generate\_pages().* (Rysunek 10.)

Obraz zawierający tekst, zrzut ekranu, Czcionka, linia

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, Czcionka, zrzut ekranu

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

Funkcja *simulation\_pages\_replacement()* uruchamia każdy z algorytmów dla każdego testu. Zapisuje liczbę błędów do pliku. Zwraca liczbę błędów dla poszczególnych algorytmów.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Rysunek

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

Rysunek

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

Rysunek

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

Rysunek

1. Wnioski

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