Ćwiczenia laboratoryjne z Systemów Operacyjnych

Symulacyjne porównanie algorytmów planowania i wymiany stron w systemie operacyjnym

Hubert Wiśniewski
środa 17:05

25 stycznia 2023

Spis treści

1 Wprowadzenie **1**

2 Symulacja planisty **2**
2.1 Parametry domyślne 2
2.2 Rozkład normalny 3
2.3 Algorytm optymalny 4

3 Symulacja wymiany stron **5**
3.1 Parametry domyślne 5
3.2 Rozkład normalny, więcej rozmiarów pamięci 6

4 Wnioski **7**
4.1 Symulacja planisty 8
4.2 Symulacja wymiany stron 8

1 Wprowadzenie

Celem projektu było wykonanie i przeprowadzenie symulacji planisty (schedulera) i menedżera wymiany stron posługujących się różnymi algorytmami i porównanie ich na podstawie określonych kryteriów:

- Planowanie:
  - Średni czas oczekiwania procesu na rozpoczęcie wykonywania go (waiting time)
  - Średni czas cyklu przetwarzania procesu, rozumianego jako suma czasu oczekiwania i czasu wykonywania procesu (turnaround time)
- Wymiana stron:
  - Średnia ilość brakujących stron w pamięci fizycznej/podręcznej (page faults)

W tym celu zaimplementowałem następujące algorytmy:

- Planowanie (bez wywłaszczania):
  - First Come First Serve
  - Shortest Job First
  - Algorytm loteryjny
  - Nieprzyczynowy algorytm optymalny minimalizujący czas cyklu przetwarzania[1]

  Algorytmy te mają na celu dokonanie wyboru następnego procesu do uruchomienia.
- Wymiana stron:
  - First In First Out (bez modyfikacji)
  - Least Frequently Used
  - Most Frequently Used
  - Nieprzyczynowy algorytm optymalny minimalizujący ilość brakujących stron

  Algorytmy te mają na celu dokonanie wyboru strony, którą należy usunąć z pamięci fizycznej/podręcznej w celu umieszczenia w niej innej żądanej strony.

2 Symulacja planisty

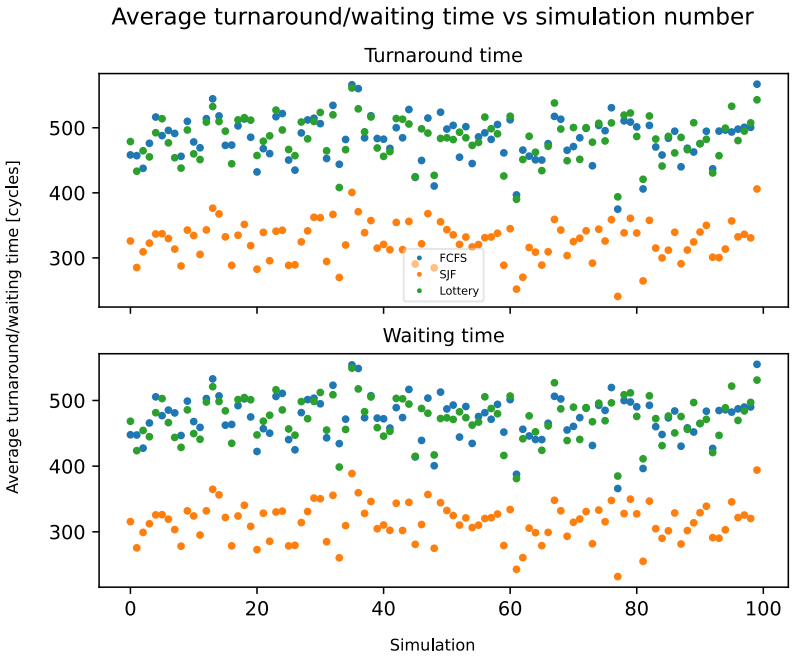
2.1 Parametry domyślne

Za pomocą skryptu `scheduler_generator.py` wygenerowałem zestaw danych wejściowych `examples/sched_in1.json`. Skorzystałem z domyślnych ustawień generatora, które zakładają utworzenie 100 symulacji po 100 procesów każda z czasem przyjścia w przedziale [0, 99] (rozkład jednostajny) i czasem trwania w przedziale [1, 20] (rozkład jednostajny). Następnie przy pomocy skryptu `scheduler_simulation.py` przeprowadziłem symulację, której wyniki są następujące:

Tabela 1: Globalnie uśrednione wyniki algorytmów, plik `examples/sched_in1.json`

Algorytm	Średni czas oczekiwania	Średni czas cyklu przetwarzania
FCFS	473.2728	483.8258
SJF	315.0275	325.5805
Loteryjny	472.3829	482.9359

[1] Algorytm optymalny został zaimplementowany zmodyfikowaną metodą brute-force, której złożoność obliczeniowa nie pozwala na przeprowadzenie symulacji o założonej długości w rozsądnym czasie. Zostanie on omówiony jedynie na przykładzie symulacji o długości 10 procesów.



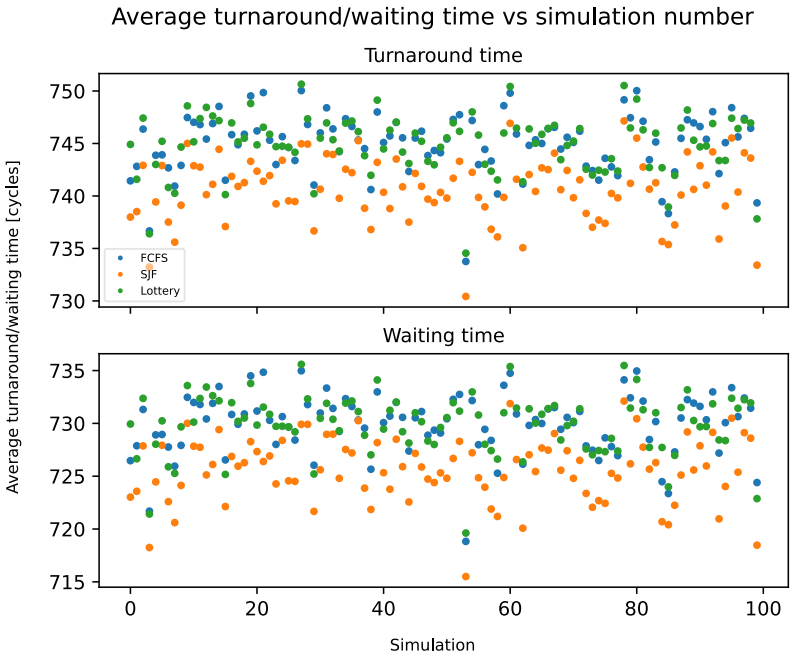
Rysunek 1: Wykres wyników symulacji z pliku `examples/sched_in1.json`

2.2 Rozkład normalny

Drugą symulację wykonałem przy użyciu danych z pliku `examples/sched_in2.json`. Wygenerowałem je za pomocą tego samego skryptu co poprzednio, jednakże dodając opcje `-f 50 5 -g 15 0.3` (rozkład czasu przyjścia normalny: $\mu = 50, \sigma = 5$, rozkład czasu trwania normalny: $\mu = 15, \sigma = 0.3$). W ten sposób zasymulowałem natłok procesów o zbliżonym do siebie czasie trwania. Uzyskałem następujące wyniki:

Tabela 2: Globalnie uśrednione wyniki algorytmów, plik `examples/sched_in2.json`

Algorytm	Średni czas oczekiwania	Średni czas cyklu przetwarzania
FCFS	729.934	744.9331
SJF	725.639	740.6381
Loteryjny	729.9089	744.908



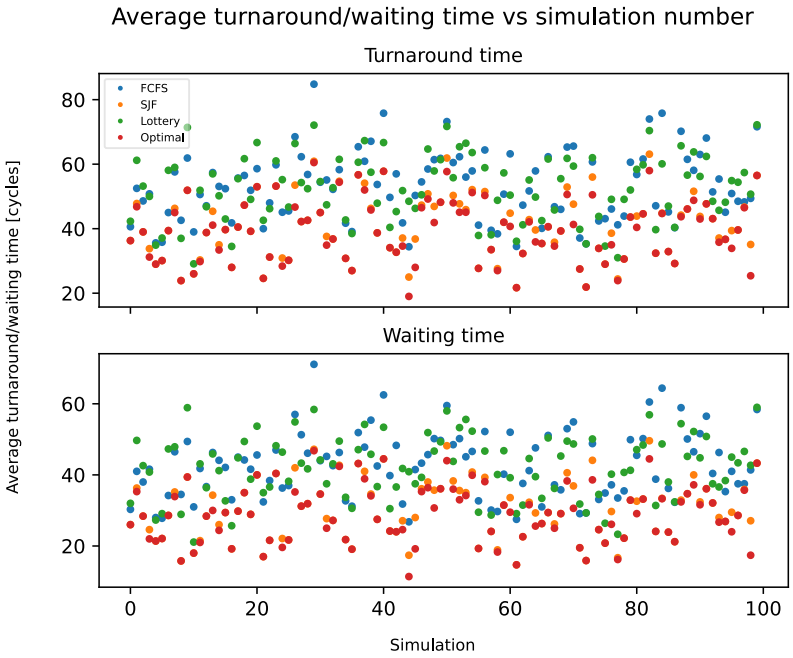
Rysunek 2: Wykres wyników symulacji z pliku `examples/sched_in2.json`

2.3 Algorytm optymalny

Aby pokazać działanie algorytmu optymalnego, wygenerowałem zbiór mniejszych symulacji `examples/sched_in3.json`. Użyłem opcji `-l 10 -c 0 9` (długość symulacji: 10 procesów, czas przyjścia w zakresie [0, 9]). Do programu symulującego dodałem opcję `-o`, która włącza algorytm optymalny. Wyniki są następujące:

Tabela 3: Globalnie uśrednione wyniki algorytmów, plik `examples/sched_in3.json`

Algorytm	Średni czas oczekiwania	Średni czas cyklu przetwarzania
FCFS	42.828	53.164
SJF	30.178	40.514
Loteryjny	42.096	52.432
Optymalny	28.965	39.301



Rysunek 3: Wykres wyników symulacji z pliku `examples/sched_in3.json`

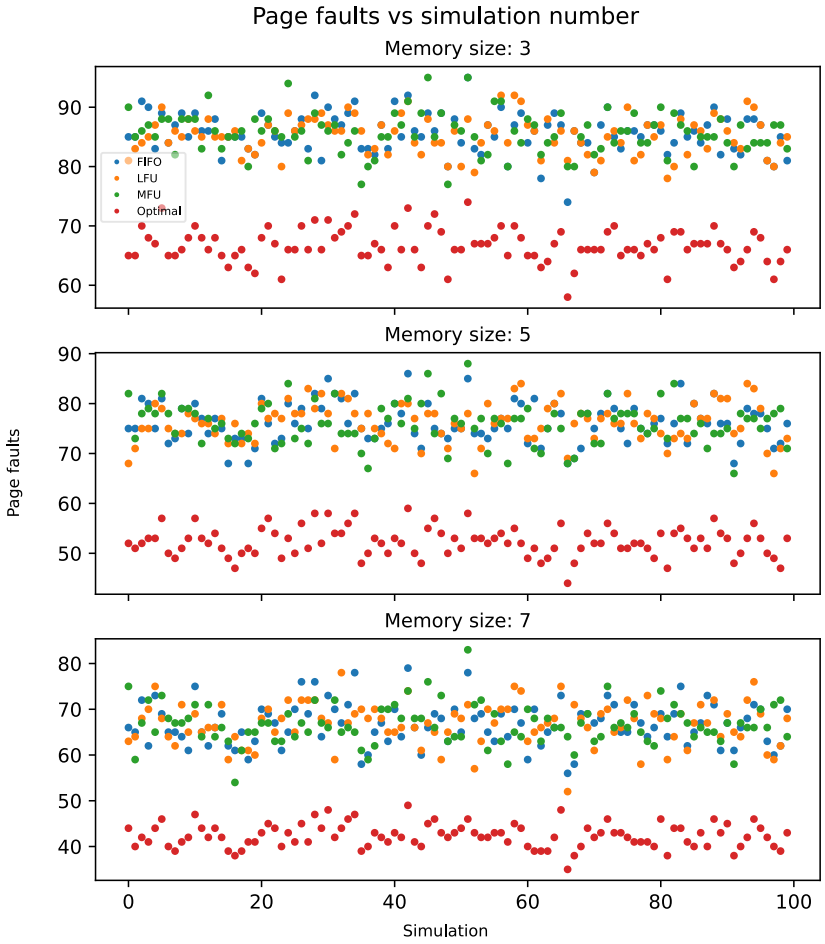
3 Symulacja wymiany stron

3.1 Parametry domyślne

Za pomocą skryptu `paging_generator.py` wygenerowałem zestaw danych wejściowych `examples/paging_in1.json`. Skorzystałem z domyślnych ustawień generatora, które zakładają utworzenie 100 symulacji po 100 żądań każda z numerem strony w przedziale [0, 19] (rozkład jednostajny) przy rozmiarach pamięci 3, 5 i 7. Następnie przy pomocy skryptu `paging_simulation.py` przeprowadziłem symulację, której wyniki są następujące:

Tabela 4: Globalna średnia ilość brakujących stron, plik `examples/paging_in1.json`

Algorytm	Rozmiar pamięci		
	3	5	7
FIFO	85.65	76.12	66.79
LFU	85.22	76.19	66.85
MFU	85.59	75.95	66.66
Optymalny	66.66	52.29	42.38



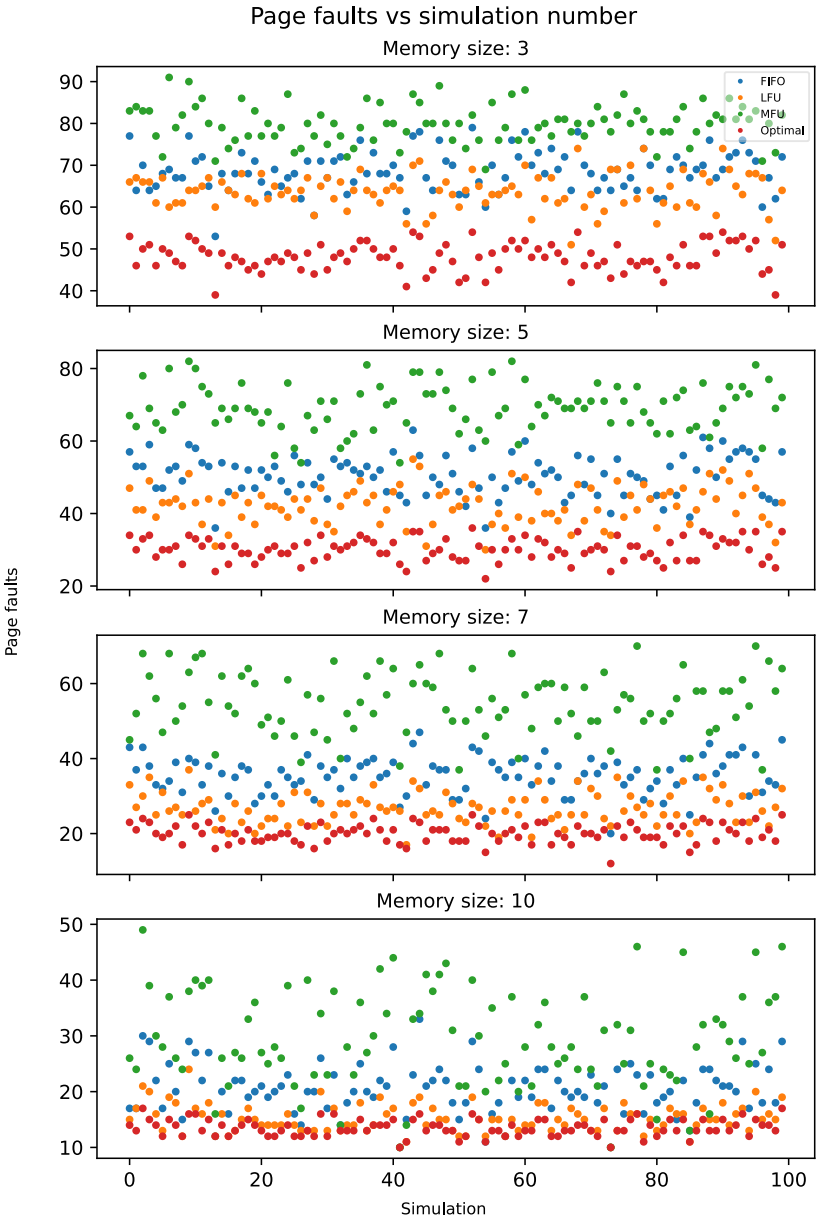
Rysunek 4: Wykres wyników symulacji z pliku `examples/paging_in1.json`

3.2 Rozkład normalny, więcej rozmiarów pamięci

W celu zasymulowania bardziej realistycznych warunków i zbadania zachowania algorytmów dla nieco większego rozmiaru pamięci, wygenerowałem plik `examples/paging_in2.json` używając opcji `-g 10 2.5 -m 3 5 7 10` (rozkład numeru żądanej strony normalny: $\mu = 10, \sigma = 2.5$, rozmiary pamięci: 3, 5, 7 i 10). Uzyskałem następujące wyniki:

Tabela 5: Globalna średnia ilość brakujących stron, plik `examples/paging_in2.json`

Algorytm	Rozmiar pamięci			
	3	5	7	10
FIFO	68.58	50.57	35.55	20.49
LFU	63.75	42.33	26.81	15.19
MFU	79.88	69.29	54.58	29.42
Optymalny	48.02	30.07	20.07	13.51



Rysunek 5: Wykres wyników symulacji z pliku `examples/paging_in2.json`

4 Wnioski

Porównania efektywności algorytmów dokonałem na podstawie kryteriów oceny przedstawionych we wprowadzeniu. Algorytm jest tym bardziej efektywny, im jego wyniki są liczbowo mniejsze (czas cyklu przetwarzania dla algorytmu planowania i ilość brakujących stron dla algorytmu wymiany stron). Optymalność jest rozumiana jako największa możliwa efektywność.

4.1 Symulacja planisty

Z przeprowadzonych symulacji wynika, że w przypadku gdy zarówno rozkład czasu przyjścia procesu, jak i czasu jego trwania jest jednostajny, zdecydowanie najlepsze wyniki spośród zaimplementowanych algorytmów (pomijając optymalny) osiąga SJF. Podobne wnioski można wyprowadzić na podstawie wyników drugiej symulacji, jednakże przewaga SJF jest w tym przypadku mniejsza. Wynika to z faktu, że w drugim zestawie danych procesy mają zbliżone czasy trwania (średnio 15 z odchyleniem standardowym 0.3) i czasy przyjścia (średnio 50 z odchyleniem standardowym 5), więc kolejność ich wykonywania nie jest aż tak istotna z punktu widzenia średniego czasu oczekiwania i cyklu przetwarzania, choć różnice między algorytmami są wykrywalne.

Niestety implementacja algorytmu optymalnego nie pozwala na wykonanie obliczeń na tak dużym zbiorze elementów w rozsądnym czasie, dlatego też przygotowałem mniejsze zestawy danych. Pomimo ponownego zastosowania rozkładu jednostajnego dla obu parametrów procesów, różnice pomiędzy wynikami algorytmów nie są tak duże, jak w przypadku pierwszej symulacji. Mniejsza ilość procesów powoduje mniejszą różnorodność wśród rozwiązań wybieranych przez algorytmy i co za tym idzie – mniejszą różnorodność wyników. Niemniej jednak można zauważyć, że algorytm SJF daje wyniki najbardziej zbliżone do optymalnych.

Z przedstawionych przykładów wynika, że algorytm SJF jest najbardziej efektywnym przyczynowym algorytmem spośród zaimplementowanych, przynajmniej dla przetestowanych przypadków. Warto jednak zwrócić uwagę, że jest on trudno stosowalny w systemach, w których czas wykonania zadania nie jest z góry określony.

Jeśli chodzi o porównanie algorytmu FCFS i loteryjnego, ten drugi wydaje się dawać średnio lepsze wyniki, choć różnica ta jest marginalna. Wygląda na to, że w przetestowanych przypadkach wybór pierwszego oczekującego procesu jest równie dobry, co wybór losowego z nich. Może być to skutkiem tego, że dane wejściowe generowane były losowo.

4.2 Symulacja wymiany stron

Z wyników pierwszej symulacji wywnioskować można, że wszystkie zaimplementowane algorytmy wymiany stron są porównywalne i żaden z nich nie daje rezultatów nawet zbliżonych do optymalnych, niezależnie od rozmiaru pamięci. Przyczyną takich wyników jest zastosowanie jednostajnego rozkładu numeru żądanej strony, co oznacza, że żądanie każdej ze stron jest równie prawdopodobne. W takiej sytuacji nie ma dobrej przyczynowej metody pozwalającej wybrać, którą stronę należy odrzucić.

Tezę tę potwierdzają wyniki drugiej symulacji, w której zastosowany został rozkład normalny o średniej wartości 10 i odchyleniu standardowym 2.5. Jako że niektóre strony są używane znacznie częściej od innych, algorytm odrzucający strony najrzadziej używane będzie dawał najlepsze wyniki. Algorytmem tym jest LFU, którego działanie opiera się właśnie na tej zasadzie. Po przeciwnej stronie skali wyników można umieścić algorytm MFU, który odrzuca najczęściej odrzucane strony, co prowadzi do znacznego przyrostu ilości brakujących stron. Algorytm FIFO wydaje się być rozwiązaniem pośrednim, jednakże dla większych rozmiarów pamięci można zauważyć, że daje on wyniki zdecydowanie bardziej zbliżone do wyników LFU niż MFU. FIFO odrzuca pierwszą stronę, która znalazła się w pamięci, ignorując tym samym rozkład prawdopodobieństwa numeru żądanej strony, co czyni go znacznie bardziej efektywnym niż MFU, który w przypadku rozkładu normalnego jest kontrproduktywny.

Algorytm LFU może zostać zastosowany z powodzeniem w większości sytuacji, gdyż jedyne czego wymaga, to zliczanie ilości żądań dostępu do danej strony.

Choć w świetle testowanych przypadków algorytm MFU może wydawać się absurdalny, to może on mieć zastosowanie praktyczne np. w buforach danych, gdzie strony najczęściej używane najprawdopodobniej już zostały przetworzone lub przesłane dalej, więc można je odrzucić.