Jednostka 10 - Paradygmat współbieżny i równoległy

Joanna Dagil

Grupa TCH-1

29 listopada 2025

1 Cel ćwiczenia

Celem ćwiczenia jest zapoznanie się z programowaniem współbieżnym i równoległym i różnicami między nimi. Wykorzystujemy wątki, procesy i różne mechanizmy synchronizacji - zamki, kolejki multiprocesowe. Rozwiązujemy też typowe problemy dla współbieżnych programów - wyścigi danych, komunikacje.

2 Zadania

Zadanie 1

Posłużyłam się kodem z tutorialu "Przetwarzanie danych w wielu wątkach", dodając `lock` na zmienną `total`. Umożliwiło to ograniczenie dostępu do niej jednemu wątkowi naraz.

```python
import threading
import time

def przetwarzaj_fragment(dane, start, end):
    global total
    fragment = dane[start:end]
    wynik = sum(fragment)
    with lock:
        total += wynik

if __name__ == "__main__":
    dane = list(range(1_000_000))
    n = len(dane)//4
    total = 0
    lock = threading.Lock()
    watki = []

    time_start = time.time()

    for i in range(4):
        t = threading.Thread(target=przetwarzaj_fragment, args=(dane, i*n, (i+1)*n))
        watki.append(t)
        t.start()

    for t in watki:
        t.join()

    time_end = time.time()

    print("Suma wszystkich elementów:", total)
    print("Czas wykonania:", time_end - time_start)
```

Wyniki:

```
Suma wszystkich elementów: 499999500000
Czas wykonania: 0.017713546752929688
```

Zadanie 2

Mechanizmem synchronizacji w tym zadaniu jest obiekt `multiprocessing.Queue`, dzięki, któremu nie dojdzie do pominięcia, żadnej części wyniku.

```
from multiprocessing import Process, Queue, cpu_count
import time

def przetwarzaj(dane, start, end, totals):
    fragment = dane[start:end]
    wynik = sum(fragment)
    totals.put(wynik)

if __name__ == "__main__":
    dane = list(range(1_000_000))
    count = cpu_count()
    n = len(dane) // count
    totals = Queue()
    total = 0
    procesy = []

    time_start = time.time()

    for i in range(count):
        p = Process(target=przetwarzaj, args=(dane, i*n, (i+1)*n if i < count-1 else
    len(dane), totals))
        procesy.append(p)
        p.start()

    for p in procesy:
        p.join()

    while not totals.empty():
        total += totals.get()

    time_end = time.time()

    print("Suma wszystkich elementów:", total)
    print("Czas wykonania:", time_end - time_start)
```

Wyniki:

```
Suma wszystkich elementów: 499999500000
Czas wykonania: 1.036818504333496
```

Zadanie 3

```
from multiprocessing import Process, Queue, cpu_count
import random

def client(id, work_queue, res_queue, count=2):
    for i in range(count):
        work = random.randint(1, 10)
        print(f"Client {id} processing work: {work}")
        work_queue.put((id, work))

        res_id, work, res = res_queue.get()
        print(f"Client {id} received result: {res} for work: {work}")

    print(f"Client {id} finished.")

def server(work_queue, res_queue):
    while True:
        item = work_queue.get()
        if item is None: # checking if signaled to stop
            break
        client_id, work = item
        print(f"Server processing work: {work} from client {client_id}")
        result = work * work  # Example processing
        res_queue[client_id].put((client_id, work, result))
```

```
24
25 if __name__ == "__main__":
26     count = cpu_count()
27     work_queue = Queue()
28     res_queue = [Queue() for _ in range(count-1)]
29
30     p_server = Process(target=server, args=(work_queue,res_queue))
31     p_server.start()
32
33     clients = []
34     for i in range(count-1):
35         p = Process(target=client, args=(i, work_queue, res_queue[i]))
36         clients.append(p)
37         print("Starting client", i)
38         p.start()
39
40     for p in clients:
41         p.join()
42
43     work_queue.put(None)  # Signal the server to stop
44     p_server.join()
45
46     print("All clients have finished.")
```

Przykładowy wynik:

```
1  Starting client 0
2  Starting client 1
3  Starting client 2
4  Starting client 3
5  Starting client 4
6  Starting client 5
7  Starting client 6
8  Client 0 processing work: 8
9  Client 2 processing work: 9
10 Client 4 processing work: 10
11 Client 3 processing work: 1
12 Server processing work: 8 from client 0
13 Server processing work: 10 from client 4
14 Client 0 received result: 64 for work: 8
15 Client 0 processing work: 4
16 Server processing work: 9 from client 2
17 Client 4 received result: 100 for work: 10
18 Client 4 processing work: 10
19 Server processing work: 1 from client 3
20 Client 2 received result: 81 for work: 9
21 Client 2 processing work: 8
22 Server processing work: 4 from client 0
23 Client 3 received result: 1 for work: 1
24 Client 3 processing work: 2
25 Client 0 received result: 16 for work: 4
26 Server processing work: 10 from client 4
27 Client 0 finished.
28 Server processing work: 8 from client 2
29 Client 4 received result: 100 for work: 10
30 Client 4 finished.
31 Client 2 received result: 64 for work: 8
32 Server processing work: 2 from client 3
33 Client 2 finished.
34 Client 3 received result: 4 for work: 2
35 Client 3 finished.
36 Client 5 processing work: 7
37 Client 6 processing work: 6
38 Server processing work: 7 from client 5
39 Server processing work: 6 from client 6
40 Client 5 received result: 49 for work: 7
41 Client 5 processing work: 3
42 Server processing work: 3 from client 5
43 Client 6 received result: 36 for work: 6
44 Client 5 received result: 9 for work: 3
45 Client 5 finished.
46 Client 6 processing work: 4
```

```
Server processing work: 4 from client 6
Client 6 received result: 16 for work: 4
Client 6 finished.
Client 1 processing work: 1
Server processing work: 1 from client 1
Client 1 received result: 1 for work: 1
Client 1 processing work: 4
Server processing work: 4 from client 1
Client 1 received result: 16 for work: 4
Client 1 finished.
All clients have finished.
```

3 Wnioski

3.1 Porównanie czasów wykonania dla zadania 1 i 2

Dla wersji wątkowej:

```
Czas wykonania: 0.017713546752929688
```

i wersji procesowej:

```
Czas wykonania: 1.036818504333496
```

W tym konkretnym przypadku wątki okazały się znacznie szybsze. Wynika to zapewne z tego, że uruchomienie procesów ma dużo większy narzut, zapewne też posługiwanie się `multiprocessing.Queue` jest kosztowne niż mechanizmem `lock`. Samo zadanie było stosunkowo krótkie, więc narzut związany z procesami był większy niż korzyści obliczeniowe z nich.

3.2 Potencjalne problemy i ich rozwiązania

3.2.1 Race condition

Przy współdzieleniu zmiennej `total` między wątkami konieczne było użycie `Lock`, aby tylko jeden wątek naraz mógł ją modyfikować. Bez tego wynik sumy byłby losowy.

3.2.2 Synchronizacja procesów

W wersji procesowej suma częściowych wyników nie mogła być trzymana we wspólnej zmiennej globalnej, dlatego użyta została `multiprocessing.Queue`, do której każdy proces odkłada swój wynik, a proces główny je sumuje.

3.2.3 Zagłodzenie

W żadnych z naszych zadań nie mogło dojść do zagłodzenia procesu, gdyż wszystkie wątki miały konkretną pracę do wykonania i po tym czasie już nie konkurował o procesor.

3.2.4 Komunikacja klient–serwer

W zadaniu 3 serwer odbierał zlecenia od wielu klientów przez wspólną kolejkę zadań i odsyłał wyniki przez osobne kolejki odpowiedzi przypisane do konkretnych klientów. Rozwiązanie to pokazuje, jak bezpiecznie wymieniać dane między procesami bez współdzielonej pamięci.