

UNIwersytet Pedagogiczny im. KEN w KRAKOWIE
Instytut Informatyki



SPRAWOZDANIE systemy czasu rzeczywistego

III rok Informatyka studia niestacjonarne

15 czerwca 2020

Spis treści

1	Temat projektu	3
2	Zespół projektowy	3
3	Cel i zakres projektu	3
4	Przypadki użycia oprogramowania	3
5	Podsumowanie i wnioski	8
6	Bibliografia	8

1 Temat projektu

Problem producenta i konsumenta w dodawaniu wiadomości do pojedynczej kolejki

2 Zespół projektowy

Szymon Grzesiak - sg.szymon8@gmail.com, Mateusz Dudziak – dudz.mateusz@gmail.com

3 Cel i zakres projektu

1. Celem projektu jest przedstawienie użycia rozwiązania programistycznego przedstawiającego klasyczny informatyczny problem synchronizacji typu producent - konsument. Zadaniem producenta jest wytworzenie produktu, umieszczenie w buforze i zaczęcie pracy od linii startu. W tym samym momencie zadaniem konsumenta jest pobranie produktu z bufora. Problemem tego zjawiska jest synchronizacja procesów, aby producent nie dodawał nowych jednostek, gdy bufor jest pełny, a konsument nie pobierał, gdy bufor jest pusty.
2. Zakresem projektu jest przedstawienie rozwiązania kolejki w której mamy nadchodzące wiadomości (z założenia liczby losowo wybrane od 0-1000) z pewną szybkością dodawane są do pojedynczej kolejki. Każdy wątek konsumencji będzie stale pobierać i przetwarzać elementy z kolejki. Ma to na celu wykazanie kompromisów między rozmiarem kolejki a liczbą nadchodzących wiadomości podczas jej przetwarzania.

4 Przypadki użycia oprogramowania

Oprogramowanie może zostać wykorzystane jako wzór obrazujący techniczne i praktyczne rozwiązanie problemu producenta i konsumenta. Oprogramowanie może zostać wykorzystane do weryfikacji szybkości nadesłanych danych opierając się na szybkości naszego procesora. Poniżej omówimy kod który został wdrożony do projektu.

Na początku importujemy wymagane moduły do realizacji projektu

```
# Ta biblioteka jest przeznaczona do wielowątkowości
import threading
import queue #implementuje 3 kolejki dla producenta i konsumenta. LIFO,FIFO,Priorytetowa
#Pozwala rejestrować zdarzenia DEBUG INFO WARNING ERROR CRITICAL
import logging
#Pomocne moduły
import time #importujemy czas
import random #importujemy moduł do losowania liczb
import string
import hashlib #zaszyfrowujemy dane, zwracay wszystkie zaszyfrowane dane w postaci prostego interfejsu
```

deklarujemy producenta i konsumenta do modułu logging

```
logging.basicConfig(level=logging.DEBUG) #zbieraj zdarzenia wyłącznie z poziomu debugowania
logger = logging.getLogger("main") #miejsce pobierania wszystkich zdarzeń
logger_p = logging.getLogger("Producent")
logger_c = logging.getLogger("Konsument")
```

Tworzymy klasę podstawową, a zarazem główną głowę naszego programu

```

class SimBase(threading.Thread):
#definiujemy główną funkcję klasy , funkcję inicjującą zachowania
    def __init__(self, q, rate):
        #ustawiamy parametry dla naszych zmiennych
        self.running = True
        self.q = q
        self.rate = rate
        self.count = 0
        #zwracamy obiekt proxy (tymczasowy obiekt nadklasy)
        super(SimBase, self).__init__()

```

Tworzymy klasę producenta

```

class SimProducer(SimBase):
    def __init__(self, q, rate):
        super(SimProducer, self).__init__(q, rate)

    def _generate_message(self):
        #zwraca losową wartość od 0 do 100
        return str(random.randint(0, 1000))

    def run(self):
        while RUNNING:
            message = self._generate_message()
            #wchodzimy do pętli jeśli bufor nie jest pełny
            if not self.q.full():
                #informuje nas o wiadomość odośnie aktualnie przetwarzanego wątku, oraz
                #generujemy wiadomość w postaci wylosowanej liczby od 0 do 1000
                logger_p.info("{} Producing {}".format(threading.current_thread().name, message))
                #wstawiamy wartość do bufora
                self.q.put(message)
                #przetwarza aktualny wątek i dodaje wylosowaną wartość do kolejki
                logger_p.debug("{} Q = {}".format(threading.current_thread().name, self.q.queue))
                #zwiększa wartość bufora o 1(do max 10)
                self.count += 1
            else:
                #w przypadku gdy kolejka jest pełna, pierwsza dodana wartość jest usuwana
                logger_p.warn("{} Q Full! Dropping {}".format(threading.current_thread().name, message))

                # Czas między zdarzeniami z rozkładu Poissona jest wykładniczy
                wait = random.expovariate(self.rate)
                logger_p.debug("{} Sleeping for = {}".format(threading.current_thread().name, wait))
                time.sleep(wait)
            logger_p.debug("{} Done".format(threading.current_thread().name))

```

Tworzymy klasę dla klienta

```

class DistributedPOW(SimBase):
    def __init__(self, q, rate, difficulty):
        self.difficulty = difficulty
        super(DistributedPOW, self).__init__(q, rate)

    def _pow(self, message):

```

```

h = None
nonce = 0
while True:
    message = "{}{}".format(message, nonce )
    message = message.encode("utf-8")
    h = hashlib.sha256(message).hexdigest()
    nonce += 1
    if self.valid_prefix(h):
        break
return (h, nonce)

def _send(self, message, h, nonce):
    # Symulujemy oczekiwanie na zasób
    # Dodawanie wątków nie zwiększa przepustowości, ponieważ
    # wątki nie są podzielone na rdzenie. Jednak tutaj nie jesteśmy zablokowani
    # zużywamy dowolne zasoby, a zatem wykorzystujemy dowolny wątek obliczający klasę POW
    # ustawiamy dostęp do procesora
    wait = random.expovariate(self.rate)
    logger_c.debug("{} Sleeping for = {}".format(threading.current_thread().name, wait))
    time.sleep(wait)

def valid_prefix(self, hash):
    return hash[:self.difficulty] == "".join(['0'] * self.difficulty)

def run(self):
    while RUNNING:
        # Jeśli tutaj zablokujemy, możemy dojść do impasu
        try:
            message = self.q.get(block=False)
            logger_c.info("{} Consuming {}".format(threading.current_thread().name, message))
            (h, nonce) = self._pow(message)
            self._send(message, h, nonce)
            logger_c.info("{} Created POW ({}, {}, {})".format
                (threading.current_thread().name, message, nonce, h))
            self.count += 1

            # Według dokumentów, jeśli zwraca prawdę, to nie gwarantuje
            # get nie będzie próbował pobrać z pustej kolejki, więc po prostu
            # zignoruj to, jeśli tak się stanie
        except queue.Empty:
            pass

    logger_c.debug("{} Done".format(threading.current_thread().name))

```

Całość kompilujemy za pomocą deklaracji w funkcji (main)

```

if __name__ == "__main__":
    random.seed(3)

    BUFF_SIZE = 10
    SIM_TIME = 10.0
    difficulty = 3
    q = queue.Queue(BUFF_SIZE) #deklaracja zmiennej q dla kolejki FIFO i rozmiarze bufora 10.
    num_consumers = 10

```

```

consumers = []

# Jeśli mamy naprawdę szybkiego producenta, kolejka się przepełni
# abyśmy mogli zrobić jedną lub dwie rzeczy, musimy zwiększyć pracowników
# lub wydłużyć czas przetwarzania
producer_rate = 1#0.1
consumer_rate = 1#3.0

# inicjujemy producenta
producer = SimProducer(q, 1.0/producer_rate)
for i in range(num_consumers):
    consumers.append(DistributedPOW(q, 1.0/consumer_rate, difficulty))

# uruchamiamy producenta
producer.start()
for w in consumers:
    w.start()

# uruchamiamy procedure
start_time = time.time()
end_time = start_time + SIM_TIME
while True:
    if time.time() > end_time:
        RUNNING = False
        break
    time.sleep(1)

# czyszczenie i obliczanie tput
# Możemy tutaj wykazać, że chociaż wątki mogą się nie zwiększać
# paramilizacja w niektórych aplikacjach może zwiększyć przepustowość
# Jeśli bawimy się liczbą klientów i stawką, możemy to pokazać
producer.join()
produce_count = producer.count
consumer_count = 0
for w in consumers:
    w.join()
    consumer_count += w.count

logger.info("Tput P={ } C={ }".format(produce_count/SIM_TIME, consumer_count/SIM_TIME))

```

Opis działania

Uruchomienie programu polega na jego kompilacji za pomocą konsoli IDLE systemu Python.

```

----- RESTART: C:\Users\dudzi\Desktop\zaa\src\prod_kon.py -----
INFO:Producent:Thread-1 Tworzę 243
DEBUG:Producent:Thread-1 Q = deque(['243'])
INFO:Konsument:Thread-11 Wykorzystuję... 243
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.8980601990514876
INFO:Producent:Thread-1 Tworzę 133
DEBUG:Producent:Thread-1 Q = deque(['133'])
INFO:Konsument:Thread-9 Wykorzystuję... 133
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.4619642979192129
INFO:Producent:Thread-1 Tworzę 618
DEBUG:Producent:Thread-1 Q = deque(['618'])
INFO:Konsument:Thread-6 Wykorzystuję... 618
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.6425558519322597
INFO:Producent:Thread-1 Tworzę 594
DEBUG:Producent:Thread-1 Q = deque(['594'])
INFO:Konsument:Thread-5 Wykorzystuję... 594
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.06777453457494967
INFO:Producent:Thread-1 Tworzę 13
DEBUG:Producent:Thread-1 Q = deque(['13'])
INFO:Konsument:Thread-10 Wykorzystuję... 13
DEBUG:Producent:Thread-1 Zamrażanie dla = 2.39490215867545
INFO:Producent:Thread-1 Tworzę 480
DEBUG:Producent:Thread-1 Q = deque(['480'])
INFO:Konsument:Thread-3 Wykorzystuję... 480
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.3002325199755253
DEBUG:Konsument:Thread-8 Gotowe!

```

Rysunek 1: Uruchomienie programu projektu

Podczas uruchomienia programu dostajemy wstępna informację o debugowaniu pierwszego wątku:

```
DEBUG:Producent:Thread-1 Q = deque(['243'])
```

Następnie uruchamiany jest proces inicjacji Q dla kolejnych wątków tworzonych przez Producenta

```

# W tym momencie tworzone jest zamrożenie dla pierwszej wiadomości
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.8980601990514876
# W tym momencie tworzony jest nowy wątek INFO:Producent:Thread-1 Tworzę 133
# W tym momencie implementujemy go do tablicy jako wartość: 133
DEBUG:Producent:Thread-1 Q = deque(['243', '133'])

```

Po utworzeniu w tablicy wartości 133 dodawany jest kolejny wątek.

Procedura ta jest powtarzana , aż do momentu zapełnienie bufora.

Bufor zostaje zapełniony w momencie kiedy w tablicy zabraknie miejsca.

Miejsca może zabraknąć jeśli tworzone wątki przekrocza granicę tablicy.

Granicę tablicy możemy przekroczyć jeśli przekroczymy 10 jej element.

#Powtórka tworzenia wątków:

```

DEBUG:Producent:Thread-1 Zamrażanie dla = 0.4619642979192129
INFO:Producent:Thread-1 Tworzę 618
DEBUG:Producent:Thread-1 Q = deque(['243', '133', '618'])
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.6425558519322597
...

```

Zapełnienie tablicy bufora wartością "406" na 10 jej elemencie:

```
INFO:Producent:Thread-1 Tworzę 406
DEBUG:Producent:Thread-1 Q = deque(['243', '133', '618', '594', '13', '480', '239', '734', '856',
DEBUG:Producent:Thread-1 Zamrażanie dla = 1.0190660934356952
```

Po zapełnieniu tablicy bufora , producent przechwytuję tę informację i rozpoczyna proces wypróżniania

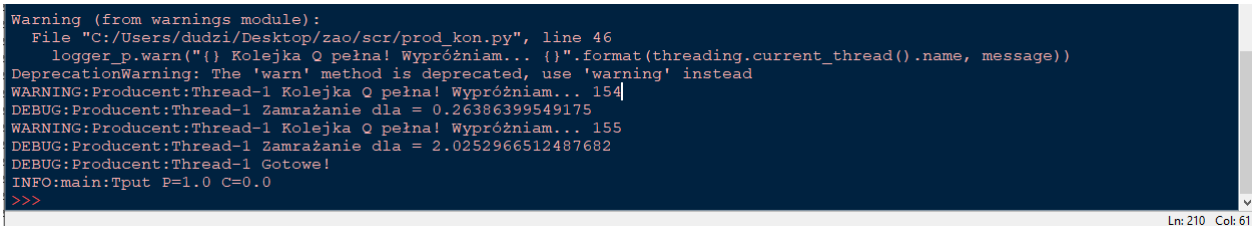
```
WARNING:Producent:Thread-1 Kolejka Q pełna! Wypróżniam... 154
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.26386399549175
WARNING:Producent:Thread-1 Kolejka Q pełna! Wypróżniam... 155
DEBUG:Producent:Thread-1 Zamrażanie dla = 2.0252966512487682
```

Aż do momentu całkowitego wyczyszczenia bufora

```
DEBUG:Producent:Thread-1 Gotowe!
```

Na koniec otrzymujemy informację czy synchronizacja producenta przebiegła pomyślnie w postaci {1}

```
INFO:main:Tput P=1.0 C=0.0
```



```
Warning (from warnings module):
  File "C:/Users/dudzi/Desktop/zao/scr/prod_kon.py", line 46
    logger_p.warn("{} Kolejka Q pełna! Wypróżniam... {}".format(threading.current_thread().name, message))
DeprecationWarning: The 'warn' method is deprecated, use 'warning' instead
WARNING:Producent:Thread-1 Kolejka Q pełna! Wypróżniam... 154
DEBUG:Producent:Thread-1 Zamrażanie dla = 0.26386399549175
WARNING:Producent:Thread-1 Kolejka Q pełna! Wypróżniam... 155
DEBUG:Producent:Thread-1 Zamrażanie dla = 2.0252966512487682
DEBUG:Producent:Thread-1 Gotowe!
INFO:main:Tput P=1.0 C=0.0
>>>
```

Rysunek 2: Program skompilowany

5 Podsumowanie i wnioski

Projekt został wykonany według stawianych wymagań. Wymagał dużo pracy pod względem zapoznania z dokumentacją problemu producenta i konsumenta jak i zapoznania z możliwością rozwiązania tego zjawiska w języku programistycznym. Wszystkie założenia zostały wdrożone , a rozwiązanie programistyczne rozwiązuje stawiany problem. Źródła z których korzystaliśmy w celu rozwiązania projektu: [1], [2], [3]. Oraz załączamy link do repozytorium z kodem na GitHubie:

<https://github.com/SzymonSG/SCR>.

6 Bibliografia

- [1] Programiz.com. *Python super() tutorial*. URL: <https://www.programiz.com/python-programming/methods/built-in/super>.
- [2] Wikipedia. *Problem producenta i konsumenta*. URL: https://pl.wikipedia.org/wiki/Problem_producenta_i_konsumenta.
- [3] Wykład z Systemów Operacyjnych; Wydział informatyki PB Wojciech Kwedło. *Synchronizacja procesów (i wątków)*. URL: <http://aragorn.pb.bialystok.pl/~wkwedlo/OS1-4.pdf?fbclid=IwAR019keDFFMJSAaWMxX4pETQGATRucOBvt7EVKhcr18Bfi5fmPjyZBj8M0>.