MATEMATYKA I INFORMATYKA NA UCZELNIACH TECHNICZNYCH

MINUT 2020 (2), s. 41–49
ISSN 2719-3063

Ewa PŁUCIENNIK[1]

[1]Wydział Automatyki, Elektroniki i Informatyki, Katedra Informatyki Stosowanej,
Politechnika Śląska, ul. Akademicka 16, 44-100 Gliwice

Semafory jako mechanizm synchronizacji procesów w systemie operacyjnym – synchronizacja procesów działających w pętlach cz. I

Streszczenie. Artykuł jest drugim, z cyklu przedstawiającego problematykę wykorzystania mechanizmu semaforów do synchronizacji procesów w systemie operacyjnym. Przedstawiono w nim proste przykłady synchronizacji procesów działających w pętlach nieskończonych. Procesy te, wypisując pojedyncze litery na konsoli rywalizują o dostęp do niej. Omówione przykłady pokazują działanie procesów bez synchronizacji oraz z synchronizacją pozwalającą uzyskać określone, dające się przewidzieć efekty. Na początku artykułu przypomniano krótko zasady funkcjonowania semaforów. Przykłady praktyczne zrealizowano z wykorzystaniem języka Python. W artykule zaproponowano również zadania do samodzielnego wykonania, bazujące na przykładach omawianych w treści artykułu.

Słowa kluczowe: proces, wątek, synchronizacja, semafor, system operacyjny, Python.

1. Wstęp

W artykule "Semafory jako mechanizm synchronizacji procesów w systemie operacyjnym - wprowadzenie" [3], który ukazał się w ubiegłym roku na łamach czasopisma MINUT, przedstawiono wprowadzenie do semaforów. Omówiono sposób ich działania, a także pokazano jak wykorzystać je do prostej synchronizacji procesów przebiegających równolegle. Zapoznanie się z ww. artykułem ułatwi zrozumienie zagadnień prezentowanych w bieżącym opracowaniu. Przypomnijmy tylko podstawowe informacje o semaforze. Semafor jest obiektem reprezentowanym przez zmienną semaforową, której nadawana jest pewna, nieujemna wartość początkowa (w dalszej treści pojęcia "semafor" i "zmienna semaforowa" będą stosowane zamiennie). Semafor posiada stowarzyszoną kolejkę, w której przebywają procesy oczekujące na dostęp do zasobu chronionego przez semafor. Nadawanie wartości początkowej semaforowi jest realizowane poza procesami, które będą korzystać z semafora, używając procedur P oraz V, opisanych poniżej. Jeżeli zdefiniujemy semafor S i nadamy mu wartość początkową *0*, oznacza to, że semafor S jest zamknięty. Ze zmienną semaforową stowarzyszone są dwie operacje, które mogą wykonać na niej procesy korzystające z semafora:

Autor korespondencyjny: E. Płuciennik (Ewa.Pluciennik@polsl.pl).
Data wpłynięcia: 16.04.2020.

- próba przejścia przez semafor tzw. procedura $P(S)$ – polega na dekrementacji (czyli zmniejszeniu o 1) zmiennej semaforowej i sprawdzeniu czy jej wartość jest mniejsza od zera; jeśli tak, to proces, który próbował przejść przez semafor zostaje zatrzymany i czeka, aż inny proces odblokuje semafor;
- odblokowanie semafora tzw. procedura $V(S)$ – polega na inkrementacji (czyli zwiększeniu o 1) zmiennej semaforowej i sprawdzeniu czy jej wartość jest większa od zera, jeśli nie, oznacza to, że jakiś proces czeka przed semaforem – wtedy oczekujący proces jest przepuszczany przez semafor.

Semafory wykorzystuje się do ochrony zasobów oraz do przekazywania informacji między procesami [2].

Niniejszy artykuł jest kontynuacją i uzupełnieniem zagadnień omawianych artykule [3]. Na podstawie prostych przykładów, przyjrzymy się jak można synchronizować procesy realizujące swoje zadania w pętli. Pokażemy jak pracują procesy pozbawione synchronizacji oraz jak wykorzystać semafory, żeby uzyskać określone efekty współdziałania procesów. Skupimy się na procesach realizujących swoje zadania pętli nieskończonej (takie procesy często są wykorzystywane w zastosowaniach przemysłowych). Rywalizację procesów o zasób zasymulujemy dostępem procesów do konsoli, na której będą one próbowały wyświetlać pojedyncze litery w pętli. Do praktycznego przeprowadzenia eksperymentów użyjemy języka Python (przydatne informacje techniczne można znaleźć w artykule [3] oraz [5], [1]). Przypomnijmy tylko, że odpowiednikiem procedury $P(S)$ w Pythonie jest metoda *acquire*(), a procedury $V(S)$ metoda *release*().

W ramach omawianych w artykule przykładów zaproponowano zagadnienia do przemyślenia i zadania do samodzielnej realizacji przez Czytelnika. Zadania te zebrano w rozdziale 5. pt."Zadania do samodzielnego wykonania". Odnoszą się one do przykładów analizowanych w treści artykułu, a bazę do ich realizacji stanowią rozwiązania omawiane w artykule. Kody użyte w artykule można pobrać pod adresem https://minut.polsl.pl/kody/kody_sem_czII.zip .

2. Rywalizacja procesów, działających w pętli, o zasoby

Jak wspomniano we wstępie, zadania realizowane przez procesy zasymulujemy wyświetlaniem pojedynczych liter. Zasobem, o który będą rywalizować procesy będzie konsola - w danym miejscu konsoli (na aktualnej pozycji kursora) można wypisać pojedynczą literę. Na początek zdefiniujemy trzy procesy, z których każdy będzie chciał wypisać, raz na każdą iterację pętli, pojedynczą literę: proces *printA*() literę A, proces *printB*() literę B, proces *printC*() literę C. Kod procesów przedstawiono poniżej.

```
def printA():
    while True:
        print('A ', end="")
        time.sleep(1)

def printB():
    while True:
        print('B ', end="")
        time.sleep(1)

def printC():
    while True:
        print('C ', end="")
        time.sleep(1)
```

Procesy będą próbowały wyświetlać swoje litery w pętli nieskończonej. Gwarantuje nam to linijka kodu o treści *while True* – definiujemy pętlę typu "wykonuj dopóki spełniony jest warunek", a naszym warunkiem jest prawda logiczna *True*. W Pythonie o zasięgu pętli, czyli instrukcjach wykonywanych w każdej jej iteracji, decyduje wcięcie w treści programu [5]. Najczęściej stosuje się wcięcie o szerokości czterech spacji. Ciało pętli obejmuje wszystkie linijki zaczynające się od wcięcia. W naszym przypadku ciało pętli *while* zawiera dwie instrukcje:

- *print('A ', end="")* – wyświetlenie litery i spacji bez przejścia do kolejnej linijki (parametr *end* funkcji *print* ma wartość pustą[1]);

[1] Użycie parametru *end* jest w funkcji *print* opcjonalne, domyślnie ma on wartość \n co oznacza przyjście do następnej linii.

- *time.sleep(1)* – zawiesza działanie danego procesu na jedną sekundę[2], zawieszenie działania procesu stosujemy po to, żeby zasymulować dodatkowe przetwarzanie w naszych procesach.

Nasze procesy będą działać równolegle. Zapewni to poniższy fragment kodu odpowiedzialny za ich uruchomienie (objaśnienia, co robią poszczególne linijki kodu zostały umieszczone w postaci komentarzy w treści kodu, dokładniejszy opis znajduje się w [3]).

```
threads = []                          # deklarujemy tablicę
threads.append(Thread(target=printA)) # dodajemy do niej procesy jako wątki
threads.append(Thread(target=printB))
threads.append(Thread(target=printC))

for thread in threads:
    thread.start()                    # uruchamiamy wątki
```

Na początku kodu naszego programu musimy zaimportować moduły umożliwiające obsługę wątków oraz usypiania. Zapewnia to nam poniższy fragment kodu.

```
from threading import Thread
import time
```

Ponieważ nasze procesy działają w pętlach nieskończonych, o zakończenie działania programu musimy zadbać sami, używając mechanizmów dostarczonych przez narzędzie, którego używamy do programowania w Pythonie (np.: w IDE Pycharm jest to kombinacja klawiszy *Ctrl + F2*).

Uruchommy nasz program, a wraz z nim trzy procesy rywalizujące o dostęp do konsoli. Na razie dostęp do naszego zasobu (konsoli) nie jest w żaden sposób regulowany. Można to porównać do grupy uczniów, z których każdy chce w konkretnym miejscu tablicy napisać jedną literę. Uda się to temu, który będzie szybszy albo potrafi się bardziej rozpychać. Przeprowadzimy teraz test tego, jaki będzie rezultat uruchomienia naszych procesów. Poniżej zaprezentowano wynik pierwszego uruchomienia.

```
A B C A C B B C A B A C B A C A B C B A C B A C
Process finished with exit code -1
```

Informacja o tym, że proces zakończył się kodem *-1* oznacza, że wykonanie procesu zastało przerwane. Jak widać na powyższym przykładzie, kolejność liter jest przypadkowa. Pytanie czy za każdym razem będzie taka sama? Uruchommy więc ponownie nasze procesy, a czynność tę powtórzmy kilkukrotnie. Przykładowe wyniki pokazano poniżej.

```
A B C B A C C A B A B C C A B B A C B C A
Process finished with exit code -1
A B C B A C A B C B A C A C B A B C B C A
Process finished with exit code -1
A B C B A C C A B B A C C B A A C B B A C
Process finished with exit code -1
```

[2] Parametrem funkcji *sleep* jest liczba sekund, parametr ten może przyjąć wartość ułamkową.

Widzimy, że za każdym razem otrzymujemy inny ciąg liter na konsoli. Co więcej, nie jesteśmy w stanie przewidzieć jaki będzie. Dzieje się tak dlatego, że nasze procesy działają w dużym stopniu niezależnie od siebie. Wpływają na siebie jedynie w ten sposób, że kiedy proces chce wypisać literę, musi poczekać na zwolnienie konsoli przez inny proces. Obowiązuje zasada: kto pierwszy ten lepszy. Można jednak zauważyć, że sekwencja zawsze zaczyna się od A B C. Decyduje o tym kolejność uruchamiania procesów. Same procesy uruchamiane są sekwencyjnie w kodzie programu – tablica *thread* jest przeglądana w pętli *for*, wątki są wstawiane do tablicy na początku programu właśnie w kolejności *printA*, *printB*, *printC*. Oczywiście może się zdarzyć, że początkowa kolejność sekwencji będzie inna, ale jest to mało prawdopodobne przy takiej konstrukcji procesów jaką mamy. Drogi Czytelniku, proponuję Ci poeksperymentować z kolejnością wstawiania wątków do tablicy *threads* oraz liczbą sekund na jaką usypiane są wątki (można również w ogóle ich nie usypiać). Czy będziesz w stanie, przed uruchomieniem programu, przewidzieć wynik?

3. Synchronizacja procesów działających w pętli

Jeżeli chcemy, żeby nasze procesy współpracowały dla osiągnięcia konkretnego celu i chcemy, żeby zachowywały się w sposób przewidywalny, niezależnie od kolejności uruchamiania czy czasu uśpienia, to musimy je zsychronizować. Posłużą nam do tego oczywiście semafory[3]. Użyjemy ich do przesyłania między procesami informacji. Załóżmy, że chcielibyśmy przy każdym uruchomieniu uzyskać porządek z poniższego przykładu.

Przykład 1. $A\ B\ C\ A\ B\ C\ A\ B\ C \ldots$.

Oznacza to, że po pierwsze musimy zapewnić sobie to, że sekwencje wypisywania rozpocznie proces pierwszy *printA*(). Po wypisaniu litery A proces *printA*() powinien wysłać informacje do procesu *printB*(), że wypisał swoją literę i teraz proces *printB*() może przejąć pałeczkę, czyli wypisać literę B. Po wysłaniu komunikatu do procesu *printB*(), proces *printA*() musi wstrzymać się z wypisaniem kolejnej litery A do momentu, aż zostanie poinformowany przez proces *printC*() o wypisaniu litery C (po wypisaniu litery C sekwencja zaczyna się powtarzać). Proces *printB*() po wypisaniu litery B powinien poinformować proces *printC*(), że może wypisać literę C, a sam będzie musiał wstrzymać się z wypisaniem kolejnej litery B do momentu uzyskania pozwolenia od procesu *printA*() (litera B zawsze musi wystąpić po A). Ponieważ musimy przekazywać trzy rodzaje komunikatów (pozwoleń na wypisanie litery):

- można wypisać jedną literę A,
- można wypisać jedną literę B,
- można wypisać jedną literę C,

to musimy zdefiniować trzy semafory (każdy odpowiedzialny za przesył pojedynczej informacji (komunikatu) – patrz artykuł [3]). Przesył komunikatu przez semafor można porównać do wydania przepustki procesowi, który oczekuje na komunikat od danego semafora. Do wysyłania komunikatu będziemy używać procedury V, a do odbierania komunikatu procedury P (procedura ta gwarantuje nam, że proces oczekujący na komunikat nie będzie realizował swoich kolejnych instrukcji, aż do otrzymania odpowiedniego

[3]Drogi Czytelniku, jeśli chcesz zrozumieć dalszą część artykułu zapoznaj się z pierwszym artykułem z cyklu [3].

komunikatu). Zanim będziemy mogli zdefiniować niezbędne semafory w naszym kodzie, musimy dołączyć odpowiednie moduły. Kod odpowiedzialny za tę operację został pokazany poniżej.

```
from threading import Semaphore, Thread
import time
```

Teraz możemy przystąpić do zdefiniowania trzech potrzebnych nam semaforów, z wykorzystaniem metody *Semaphore*(), której parametrem jest wartość początkowa zmiennej semaforowej.

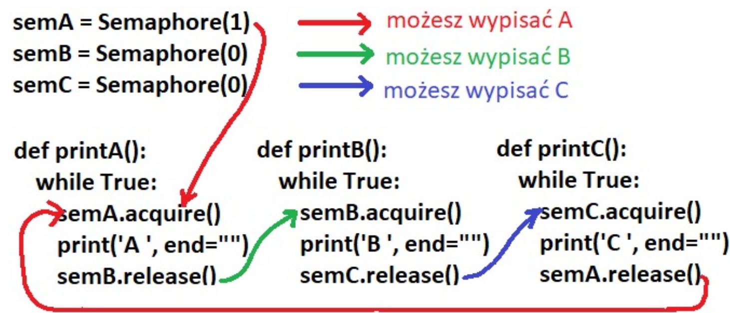
```
semA = Semaphore(1)
semB = Semaphore(0)
semC = Semaphore(0)
```

Semafor *semA* będzie odpowiedzialny za przekazanie komunikatu "można wypisać pojedynczą literę *A*", semafory *semB* i *semC* będą odpowiedzialne, odpowiednio za komunikaty dla liter *B* i *C*. Zwróćmy uwagę, że deklaracja semafora *semA* różni się nieco, od deklaracji pozostałych semaforów. Semafory *semB* i *semC* mają początkową wartość zmiennej semaforowej równą *0* (która oznacza, że nie można wypisać litery *B* ani *C*), natomiast dla semafora *semA* wynosi ona *1*, ponieważ musimy rozpocząć wypisywanie sekwencji od litery *A*. Zanim uzupełnimy kod naszych procesów o wywołania procedur *P* i *V* niezbędnych do nadawania i odbierania komunikatów przypomnijmy ponownie, że w języku Python jak i w wielu innych, odpowiednikiem procedury *P* jest metoda *acquire*(), a procedury *V* metoda *release*(). Przyjrzyjmy się teraz jak zdefiniowane semafory zostały użyte w poniższym kodzie.

```
def printA():                def printB():                def printC():
    while True:                  while True:                  while True:
        semA.acquire()               semB.acquire()               semC.acquire()
        print('A ', end="")          print('B ', end="")          print('C ', end="")
        time.sleep(1)                time.sleep(1)                time.sleep(1)
        semB.release()               semC.release()               semA.release()
```

Zauważmy, że wypisanie każdej z liter, w pojedynczej iteracji pętli *while* jest poprzedzone wywołaniem metody *acquire*() (procedury *P*) na odpowiednim semaforze, czyli oczekiwaniem na komunikat, że można daną literę wypisać. Po wypisaniu litery, każdy z procesów wykonuje metodę *release*() (procedurę *V)*, wysyłając komunikat, że można wypisać kolejną literę w sekwencji. I tak, proces *printA*() po wypisaniu litery *A* nadaje komunikat za pomocą semafora *semB*, wywołując metodę *release*(), że można już wypisać literę *B*. Na ten komunikat czeka proces *printB*() zatrzymany przez metodę *acquire*() wywołaną na semaforze *semB*. Po przejściu przez semafor *semB* (odebraniu komunikatu z pozwoleniem) proces *printB*() wypisuje literę *B* i nadaje komunikat przy pomocy metody *release*() wywołanej na semaforze *semC*, że można wypisać literę *C*. Proces *printC*() oczekuje na pozwolenie wypisania litery *C*, wywołując metodę *acquire*() na semaforze *semC*. Po otrzymaniu pozwolenia, wypisuje literę, a następnie wysyła informację, że można rozpocząć sekwencję od nowa (czyli od litery *A*), wywołując metodę *release*() na semaforze *semA*. Zwróćmy jeszcze raz uwagę na to, że proces *printA*(), jako ten, który ma rozpocząć sekwencję musi dostać pozwolenie na to, wynikające z wartości początkowej semafora *semA* wynoszącej *1*. Pozwolenie to wykorzystuje w pierwszej iteracji pętli. W kolejnych iteracjach musi czekać na pozwolenie od procesu *printC*(). Graficznie przekazywanie komunikatów w naszym kodzie pokazuje rysunek 1 (nie uwzględniono na nim funkcji *sleep*).

Nasze procesy są zsynchronizowane i niezależnie od okoliczności (zmiany kolejności procesów w tabeli *threads* lub czasów usypiania procesów) przy każdym uruchomieniu wypisana zostanie ta sama sekwencja



Rysunek 1. Przesył komunikatów między procesami wypisującymi sekwencję ABC.

– Drogi Czytelniku sprawdź to oczywiście sam. Spróbuj również zmienić wyświetlaną sekwencję, jak to podano w zadaniu 1.

Synchronizacja naszych procesów, w celu uzyskania powtarzających się raz na sekwencję liter jest stosunkowo prosta. Skomplikujmy zatem nasze zadanie. Powiedzmy, że chcemy uzyskać ciąg z poniższego przykładu.

Przykład 2. $A\ A\ B\ C\ A\ A\ B\ C\ldots$

W powtarzającej się w nim sekwencji mamy najpierw dwie litery A, a potem pojedyncze B i C. Spróbujemy przy pomocy, tylko i włącznie, semaforów zmusić nasze procesy do zmiany sekwencji (nie będziemy stosować modyfikacji kodu polegającej na dopisaniu kolejnej litery A w funkcji *print('A ', end="")*). Na początek, ponieważ nasza sekwencja ma zaczynać się od dwóch liter A, zmienimy wartość semafora *semA* na liczbę *2* – oznacza to wydanie dwóch pozwoleń na wyświetlenie litery A.

```
semA = Semaphore(2)
```

Powyższa zmiana w kodzie spowodowała, że nie uzyskaliśmy oczekiwanej sekwencji, a co więcej wynik jego uruchomienia znowu staje się, nie do końca przewidywalny. Wybrane trzy wyniki wielokrotnego uruchomienia zmodyfikowanego kodu przedstawiono poniżej. Drogi Czytelniku, w ramach analizy zachowania semaforów w tym przykładzie, spróbuj wykonać zadanie 2.

```
A A B B C A C A B B C A C B A B C C A A B C B
Process finished with exit code -1
A A B C B C A A B C B A C B A C B A C B A C B
Process finished with exit code -1
A B A C B A C B A C B A C B A C B A C B A C B
Process finished with exit code -1
```

Nawet na podstawie tych trzech wyników można zauważyć pewną prawidłowość (pamiętając, że procesy usypiane są na ten sam czas). Sekwencja rozpoczyna się zawsze od trzech liter ze zbioru $\{A, B\}$ przy czym pierwsza litera to zawsze A, potem przemieszanie liter zmienia się przy każdym uruchomieniu. Taki efekt to rezultat działania naszych semaforów. Na początek należy podkreślić to, że każde wyświetlenie litery A wiąże się z wydaniem pozwolenia na wyświetlenie litery B, a wszystkie procesy działają równolegle i mogą zostać wstrzymane (do czasu ich zwolnienia) jedynie w momencie wykonania metody *acquire*() (procedury P) na semaforze (nie licząc oczywiście funkcji *sleep*). Przejście przez semafor następuje natychmiast po pojawieniu się pozwolenia. Ponieważ na początku mamy dwa pozwolenia na

wyświetlenie litery A, proces *printA*() wykonuje natychmiastowo dwie iteracje swojej pętli, jednocześnie w pierwszej iteracji, wydając pozwolenie na wyświetlenie litery B. W pewnym momencie istnieją więc dwa pozwolenia jednocześnie: "możesz wyświetlić literę A" oraz "możesz wyświetlić literę B" co oznacza, że procesy *printA*() i *printB*() dostały zgodę na dostęp do konsoli. Ponieważ nie mogą jednocześnie z niej skorzystać, swoje pozwolenie pierwszy zrealizuje ten proces, który będzie miał więcej szczęścia (działa minimalnie szybciej). Stąd nie jesteśmy w stanie powiedzieć czy po pierwszej literze A wyświetli się druga litera A, czy pierwsza litera B. Konsekwencją tej niepewności (braku determinizmu) jest oczywiście to, że nie jesteśmy w stanie przewidzieć dalszego ciągu sekwencji.

Przeanalizujmy raz jeszcze sekwencję, którą chcemy uzyskać: $A\ A\ B\ C\ A\ A\ B\ C\ldots$. Widzimy, że na każde dwie litery A muszą być wyświetlone pojedyncze litery B i C. Żeby rozpocząć sekwencję musimy mieć na początku dwa pozwolenia na wyświetlenie litery A – wykonają się dwie iteracje pętli w procesie *printA*() i zostaną wydane dwa pozwolenia na wypisanie litery B. Zatem każde wyświetlenie litery B powinno wymagać otrzymania potwierdzenia wypisania dwóch liter A (czyli dwukrotnego przejścia przez semafor *semB*). Po wyświetleniu pojedynczej litery B następuje wydanie pozwolenia na wyświetlenie pojedynczej litery C. Natomiast każde wyświetlenie litery C musi skutkować wydaniem pozwoleń na wyświetlenie dwóch liter A – dwukrotnie trzeba wykonać metodę *release*() na semaforze *semA*. Kod procesów po modyfikacji przedstawiono poniżej.

```
semA = Semaphore(2) # dwa pozwolenia na wypisanie litery A
semB = Semaphore(0)
semC = Semaphore(0)

def printA():
    while True:
        semA.acquire()
        print('A ', end="")
        time.sleep(1)
        semB.release()

def printB():
    while True:
        semB.acquire()
        semB.acquire()
        print('B ', end="")
        time.sleep(1)
        semC.release()

def printC():
    while True:
        semC.acquire()
        print('C ', end="")
        time.sleep(1)
        semA.release()
        semA.release()
```

Synchronizacja jakiej dokonaliśmy zapewnia nam uzyskanie zadanej sekwencji niezależnie od warunków uruchomienia procesów. Na podstawie tego rozwiązania możesz, Drogi Czytelniku, wykonać zadanie 2.

Rozważmy teraz inny ciąg.

Przykład 3. $A\ B\ C\ A\ A\ B\ C\ A\ A\ B\ C\ A\ldots$

Zauważmy, że w naszym zadaniu powtarza się sekwencja $A\ B\ C\ A$. Można jednak spojrzeć na ten ciąg trochę inaczej, zauważając, że zaczyna się on od pojedynczej litery A, a potem powtarza się sekwencja $B\ C\ A\ A$ (pamiętajmy, że nasze procesy działają w pętlach nieskończonych i wyświetlanie ciągu może być przerwane w dowolnej chwili czasu). Na początek rozważmy powtarzającą się sekwencję. Żeby ją uzyskać, po wypisaniu pojedynczej litery B, proces *printB*() musi wydać jedno zezwolenie na wypisanie litery C. Na to pozwolenie czeka proces *printC*(). Po jego uzyskaniu wyświetla literę C, a następnie wydaje dwa pozwolenia na wypisanie litery A. Proces *printA*() wykorzystuje te dwa pozwolenia w dwóch iteracjach, wypisując dwie litery A i jednocześnie, wydając dwa pozwolenia na wyświetlenie litery B. Te dwa pozwolenia konsumuje proces *printB*() w pojedynczej iteracji, żeby wyświetlić literę B i cykl zaczyna się od początku. Musimy tylko pamiętać, że ciąg powinien zacząć się od pojedynczej litery A – w związku z tym

ustawiamy wartość początkową semafora *semA* na jeden. Pozwoli to rozpoczęcie wyświetlania ciągu. Należy jednak zwrócić uwagę na to, że proces *printB*() do wyświetlenia litery pojedynczej *B* potrzebuje aż dwóch pozwoleń, w związku z tym do wyświetlenia pierwszej litery *B* potrzebujemy jeszcze jednego pozwolenia – udzielamy go, nadając semaforowi *semB* wartość początkową *1*. Wyświetlenie pierwszej litery *B* w naszym ciągu wykorzysta dwa pozwolenia: jedno pochodzące z wartości początkowej semafora *semB*, drugie udzielone przez proces *printA*(). Kod procesów zamieszczono poniżej.

```
semA = Semaphore(1) # jedno pozwolenie na wypisanie początkowej litery A
semB = Semaphore(1) # jedno pozwolenie na wypisanie początkowej litery B
semC = Semaphore(0)

   def printA():             def printB():             def printC():
       while True:               while True:               while True:
           semA.acquire()            semB.acquire()            semC.acquire()
           print('A ', end="")       semB.acquire()            print('C ', end="")
           time.sleep(1)             print('B ', end="")       time.sleep(1)
           semB.release()            time.sleep(1)             semA.release()
                                     semC.release()            semA.release()
```

Drogi Czytelniku, po uruchomieniu i przeanalizowaniu powyższego kodu, spróbuj zrealizować zadanie 3.

4. Podsumowanie

W artykule (drugim z cyklu poświęconego zagadnieniu synchronizacji i semaforom) pokazano proste przykłady synchronizacji procesów działających, w sposób ciągły (w pętlach nieskończonych), równolegle w systemie operacyjnym. Omówiono konsekwencje braku synchronizacji takich procesów, w przypadku kiedy rywalizują o wspólny zasób. Przedstawiono proste programy umożliwiające praktyczne przećwiczenie synchronizacji takich procesów z wykorzystaniem semaforów. Zrozumienie przedstawionych przykładów i realizacja, zaproponowanych w artykule, zadań do samodzielnego wykonania pozwoli Czytelnikowi zrozumieć niektóre problemy synchronizacji i pogłębić wiedzę na temat mechanizmu semaforów. Pomocą przy realizacji zadań będzie artykuł wprowadzający w tematykę semaforów [3] (pierwszy z cyklu). W kolejnym artykule z cyklu [4] przyjrzymy się jak uzyskać bardziej skomplikowane sekwencje liter oraz przeanalizujemy, jak na synchronizację wpłynie ograniczenie liczby iteracji w pętlach wykonywanych przez procesy.

5. Zadania do samodzielnego wykonania

Zadanie 1. Bazując na rozwiązaniu przykładu 1, proszę spróbować uzyskać następujące sekwencje:

- $C\ B\ A\ C\ B\ A\ldots$,
- $B\ A\ C\ B\ A\ C\ldots$.

Zadanie 2. Dla przykładu 2, proszę spróbować uzyskać następujące sekwencje liter:

- $A\ A\ A\ B\ C\ A\ A\ A\ B\ C\ \ldots,$
- $A\ B\ B\ C\ A\ B\ B\ C\ \ldots,$

pamiętając, że używamy tylko mechanizmu semaforów do synchronizacji procesów i nie modyfikujemy funkcji *print*.

Zadanie 3. Na podstawie przykładu 3, proszę spróbować uzyskać następujące ciągi:

- $B\ A\ C\ B\ B\ A\ C\ B\ \ldots,$
- $C\ A\ B\ C\ C\ A\ B\ C\ \ldots.$

Proszę zwrócić szczególną uwagę na litery rozpoczynające zadane ciągi oraz wartości początkowe semaforów.

Podziękowania

Autorka pragnie podziękować recenzentom za trud włożony w recenzje.

Literatura

1. J.A. Briggs, *Python dla dzieci. Programowanie na wesoło*, PWN, Warszawa 2016.
2. A.B. Downey, *The Little Book of Semaphores*, 2016.[4]
3. E.Płuciennik *Semafory jako mechanizm synchronizacji procesów w systemie operacyjnym - wprowadzenie*, MINUT 2019 (1), s. 17-23.
4. E.Płuciennik *Semafory jako mechanizm synchronizacji procesów w systemie operacyjnym – synchronizacja procesów działających w pętlach cz. II*, MINUT 2020 (2), s. 50-62.
5. *The Python Standard Library, Synchronization Primitives*, Python Software Foundation.

[4]Darmowa książka dostępna pod adresem, np. http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf .