MINUT 2020 (2), s. 50–62
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. II

Streszczenie. Artykuł, trzeci z cyklu poruszającego problematykę synchronizacji procesów, przedstawia proste przykłady synchronizacji procesów działających równolegle, w pętlach skończonych. Synchronizacja jest realizowana z wykorzystaniem mechanizmu semaforów. W przykładach posłużono się procesami rywalizującymi o dostęp do konsoli. Pokazano przykłady procesów blokujących swoje działanie oraz rozwiązanie problemu blokady w konkretnych przypadkach. Przedstawiono również przykłady synchronizacji wymagające użycia koordynatora. Przykłady zamieszczone w artykule realizowano z wykorzystaniem języka Python.
Słowa kluczowe: proces, wątek, synchronizacja, semafor, system operacyjny, Python.

1. Wstęp

Niniejszy artykuł jest trzecim z cyklu poświęconego problemom synchronizacji. W pierwszym artykule z cyklu [3] omówiono podstawowe pojęcia związane z semaforami, przedstawiono sposób ich działania oraz pokazano, na prostych przykładach, jak przy pomocy semaforów można synchronizować pracę procesów realizowanych równolegle w systemie operacyjnym. W drugim artykule z cyklu [4] przedstawiono przykłady synchronizacji procesów równoległych realizujących swoje działanie w pętlach. Przykłady te bazowały na prostych procesach wypisujących, w pętlach nieskończonych, pojedyncze litery na konsolę, a wykorzystanie semaforów pozwoliło uzyskiwać określony porządek wyświetlanych liter. Do zrozumienia zagadnień omawianych w niniejszym artykule konieczne jest zapoznanie się z poprzednimi artykułami w cyklu, w szczególności z artykułem drugim, ponieważ przykłady synchronizacji w nim przedstawione posłużą jako baza do dalszych rozważań. Tak samo jak w poprzednich artykułach, przykłady będą realizowane z wykorzystaniem języka Python [1], [5]. Przypomnijmy, że z semaforem S stowarzyszone są dwie procedury $P(S)$ i $V(S)$. Procedura P służy procesom do zażądania dostępu do zasobu – podjęcia próby przejścia przez semafor. Wywołując procedurę V procesy informują semafor o zwolnienia zasobu,

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

z którego korzystały. Odpowiednikiem procedury $P(S)$ w Pythonie jest metoda *acquire*(), a procedury $V(S)$ metoda *release*().

Przy omawianych przykładach znajdziesz, Drogi Czytelniku, zagadnienia do przemyślenia i samodzielnej realizacji. Zadania te zebrano w rozdziale 6. pt."Zadania do samodzielnego wykonania". Odnoszą się one do przykładów analizowanych w treści artykułu, a bazą do ich realizacji są kody źródłowe procesów omawiane w artykule. Kody te można pobrać pod adresem https://minut.polsl.pl/kody/kody_sem_czIII.zip .

2. Procesy działające w pętlach skończonych

Na początek przypomnimy strukturę naszych procesów, które rywalizują o dostęp do konsoli chcąc wypisać, w każdej iteracji pętli, pojedynczą literę. Kod procesów przedstawiono poniżej.

```
def printA():                def printB():                def printC():
    global COUNTER               global COUNTER               global COUNTER
    for i in range(COUNTER):     for i in range(COUNTER):     for i in range(COUNTER):
        print('A ', end="")          print('B ', end="")          print('C ', end="")
        time.sleep(1)                time.sleep(1)                time.sleep(1)
```

Powyższy kod różni się on od tego wykorzystywanego w poprzednim artykule m.in. konstrukcją pętli. Zamiast pętli nieskończonej *while True* użyjemy pętli *for i in range*(*COUNTER*). W ramach pętli zadeklarowana została zmienna sterująca *i*, która przyjmie, kolejno wartości: *0*, *1*, *2*, ..., *COUNTER-1*[1]. Każda z pętli wykona tę samą liczbę iteracji – w wątkach użyto globalnej zmiennej *COUNTER* (deklaracja *global COUNTER*). Zmienna *COUNTER* zostanie zadeklarowana na początku programu, po instrukcjach odpowiedzialnych za włączenie do naszego programu odpowiednich bibliotek, zapewniających obsługę pracy wielowątkowej oraz możliwość usypiania procesów, jak to pokazano w poniższym fragmencie kodu. Zmienna *COUNTER* przyjmie wartość pięć, co oznacza, że w każdej pętli *for* wykona się pięć iteracji.

```
from threading import Thread     #import biblioteki obsługującej wątki
import time                      #import biblioteki pozwalającej używać funkcji sleep()

COUNTER = 5 #deklaracja zmiennej globalnej definiującej liczbę iteracji
```

Nasz program będzie się kończył instrukcjami odpowiedzialnymi za deklarację i uruchomienie wątków, pokazanymi na poniższym kodzie. Wątki zostały zebrane w tablicy *threads*. Następnie są uruchamiane poleceniem *thread.start*() w pętli *for* przeglądającej tę tablicę. Kolejna pętla *for* służy do oczekiwania na zakończenie wątków (instrukcja *thread.join*()). Kiedy wszystkie wątki zakończą swoją pracę, wypisywany jest komunikat o zakończeniu programu.

```
threads = []                           #deklaracja tablicy wątków
threads.append(Thread(target=printB))  #dodanie wątków do tablicy
threads.append(Thread(target=printA))
threads.append(Thread(target=printC))
```

[1]Funkcja *range*(*n*) zwraca domyślnie sekwencję liczb z zakresu od 0 do n-1 z krokiem 1

```
for thread in threads:
    thread.start()       #uruchomienie wątków
for thread in threads:
    thread.join()        #oczekiwanie na zakończenie wątków
print("\nAll done")      #komunikat końcowy
```

Zanim przejdziemy do kolejnych zagadnień proponuję Ci, Drogi Czytelniku, kilkukrotne uruchomienie powyższego kodu i zaobserwowanie wyników wyświetlonych na konsoli. Możesz spróbować zmienić kolejność dodawania wątków do tablicy *threads*. Przekonasz się, że za każdym razem uzyskujemy nieco inny ciąg złożony z liter A, B i C, przy czym liczba wystąpień każdej z liter jest równa wartości zmiennej *COUNTER*. Przykładowe wyniki dwukrotnego uruchomienia zaprezentowano poniżej.

```
B A C A C B C A B C B A A B C
All done
B A C A B C A C B B C A A B C
All done
```

Zwróćmy uwagę, że po każdym ciągu liter wyświetlany jest komunikat *All done*. Co więcej, Python poinformuje nas o poprawnym zakończeniu programu komunikatem: *Process finished with exit code 0*. Drogi Czytelniku, spróbuj zsynchronizować procesy tak, żeby uzyskać porządek $A\ B\ C\ A\ B\ C\ \ldots$ – skorzystaj z przykładu 1 opisanego w [4]. Nie powinno to sprawić Ci większej trudności.

Spróbujemy teraz wypróbować kolejną przykładową sekwencję liter (przykład 2) z artykułu [4]. Tym razem naszym celem będzie uzyskanie ciągu $A\ A\ B\ C\ A\ A\ B\ C\ \ldots$. Poniżej zaprezentowano kod procesów, zsynchronizowanych tak jak w przykładzie ze wspomnianego artykułu, bez uwzględnienia kodu uruchomienia wątków i oczekiwania na ich zakończenie (kod ten, zaprezentowany powyżej musimy oczywiście dołączyć do naszego programu). Zrezygnujemy również z usypiania wątków (przy naszej synchronizacji nie będzie to miało to znaczenia).

```
from threading import Thread, Semaphore #deklaracja niezbędnych bibliotek

semA = Semaphore(2) #deklaracja semaforów odpowiedzialnych za wyświetlanie
semB = Semaphore(0) #poszczególnych liter
semC = Semaphore(0)
COUNTER = 5

   def printA():               def printB():               def printC():
       global COUNTER              global COUNTER              global COUNTER
       for i in range(COUNTER):    for i in range(COUNTER):    for i in range(COUNTER):
           semA.acquire()              semB.acquire()              semC.acquire()
           print('A ', end="")         semB.acquire()              print('C ', end="")
           semB.release()              print('B ', end="")         semA.release()
                                       semC.release()              semA.release()
```

Po uruchomieniu naszego kodu wynik będzie następujący.

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

Po pierwsze możemy zaobserwować, że ciąg kończy się na pojedynczej literze *A*, zawiera pięć liter *A* i po dwie litery *B* oraz *C*. Co więcej, nie można się doczekać końca programu i trzeba zakończyć jego działanie ręcznie. Rezultat jest taki sam przy każdym uruchomieniu. Drogi Czytelniku spróbuj zaobserwować jak zmieni się wyświetlany ciąg, kiedy zmienisz wartość zmiennej globalnej *COUNTER*.

Możemy się domyślać, że z jakiegoś powodu procesy nie mogą dokończyć swojego działania. Żeby dowiedzieć się czegoś więcej, o tym co stało się w trakcie wykonywania programu, zmodyfikujemy nieco kod uruchomienia wątków. Modyfikacja ta pozwoli nam dokładniej prześledzić działanie programu. Na początek dołączymy nową bibliotekę, dopisując jej deklarację (*import sys*) do listy bibliotek używanych w naszym programie, jak to pokazano poniżej.

```
from threading import Thread, Semaphore #deklaracja niezbędnych bibliotek
import sys
```

Włączenie biblioteki *sys* do naszego programu pozwoli nam skorzystać z funkcji *exit*() umożliwiającej zakończenie działania programu.

We fragmencie kodu uruchamiającego procesy wprowadzimy kilka przydatnych zmian. Po pierwsze, w metodzie tworzącej wątki (*Thread*()), wywoływanej w poleceniu *threads.append*(), nadamy naszym wątkom nazwy (parametr *name*) w celu ich łatwiejszej identyfikacji – pamiętajmy o tym, że nazwa wątku nie jest tożsama z nazwą funkcji, która jest uruchamiana w wątku (parametr *target*). Poza tym, utworzymy nasze wątki jako tzw. demony (parametr *daemon=True*). Demon to proces działający w systemie operacyjnym, niewymagający interakcji z użytkownikiem oraz posiadający proces nadrzędny. W naszym przypadku, użycie demonów gwarantuje nam, że ich wykonywanie zostanie przerwane, jeżeli nasz program wykona instrukcję *sys.exit*(), niezależnie od tego czy zakończyły swoją pracę czy nie, ponieważ nasz program będzie dla tych wątków procesem nadrzędnym. Opisane modyfikacje pokazano w poniższym kodzie.

```
threads = []    #deklaracja tablicy wątków
threads.append(Thread(target=printA, name="threadA", daemon=True))
threads.append(Thread(target=printB, name="threadB", daemon=True))
threads.append(Thread(target=printC, name="threadC", daemon=True))
```

Kolejna modyfikacja dotyczy pętli *for*, w której oczekujemy na zakończenie wszystkich wątków. W instrukcji *thread.join*(*timeout=1*) pojawił się parametr funkcji *join*() o nazwie *timeout*. Parametr ten określa jak długo (w sekundach) należy czekać na zakończenie wątku. Po upływie określonego czasu program główny przechodzi do dalszych instrukcji. Nie oznacza to jednak, że wątki, które nie zdążą się zakończyć przed upływem tego czasu, kończą swoje działanie – dalej pracują w tle. U nas czas oczekiwania wynosi jedną sekundę – ponieważ nie zastosowaliśmy usypiania wątków, czas ten jest wystarczający, żeby zakończyły one swoje działanie. Fragment kodu odpowiedzialnego za uruchomienie wątków i oczekiwanie na ich zakończenie pokazano poniżej.

```
for thread in threads:      #uruchomienie wątków
    thread.start()
for thread in threads:      #oczekiwanie na zakończenie wątków
    thread.join(timeout=1)
```

Następnie uzupełnimy nasz kod o dodatkową pętlę *for*, w której sprawdzamy funkcją *thread.isAlive*() czy wątek nadal jest aktywny i wyświetlamy odpowiednią informację. Potem wypisujemy na ekran aktualne wartości poszczególnych semaforów (posłużyliśmy się tu specjalną zmienną *_value* dla semafora[2]).

[2]Nie we wszystkich wersja języka Python będzie to działać, przetestowane zostało na wersji 3.7.

W dalszej kolejności wypisujemy komunikat o treści *All done* oraz wywołujemy funkcję systemową *sys.exit()*, która zakończy nasz program główny, jednocześnie przerywając pracę wątków (naszych demonów), które wciąż działają w tle. Fragment nowego kodu pokazano poniżej.

```
print("\n")
for thread in threads:
    if(thread.isAlive()):
        print(thread.getName()+" is alive")
    else:
        print(thread.getName() + " has ended")
print('semA value is {}'.format(semA._value))
print('semB value is {}'.format(semB._value))
print('semC value is {}'.format(semC._value))
print("\nAll done")
sys.exit() #zakończenie programu i wszystkich wątków w nim uruchomionych
```

Tak zmodyfikowany kod gwarantuje nam, że po upływie jednej sekundy od uruchomienia wątków, wyświetlą się informacje o tym, które wątki nadal pracują, a które zdążyły się zakończyć. Następnie otrzymamy informacje o wartościach semaforów i program zakończy się, kończąc równocześnie działanie wszystkich, jeszcze aktywnych, wątków.

Wyniki uruchomienia naszego programu pokazano poniżej. Przeanalizujmy je.

```
A A B C A A B C A

threadA has ended
threadB is alive
threadC is alive
semA value is 1
semB value is 0
semC value is 0

All done

Process finished with exit code 0
```

Oczywiście modyfikacje, których dokonaliśmy nie wpłynęły na wyświetlany ciąg liter, ale dodatkowo wypisane informacje, pozwalają nam stwierdzić, że o ile proces *printA*() (a właściwie wątek odpowiedzialny za jego realizację) zakończył się, to dwa pozostałe procesy wciąż były aktywne (coś spowodowało, że nie mogły się zakończyć w określonym czasie). Ich działanie zostało przerwane dopiero instrukcją *sys.exit*(). W naszych prostych procesach jedynym miejscami, w których mogły one zostać zatrzymane są wywołania procedury P (metody *acquire*()) na semaforze, którego wartość wynosiła zero [3]. Widzimy, że semafory odpowiedzialne za wypisywanie liter B i C mają właśnie taką wartość, co oznacza, że procesy *printB()* i *printC()* zostały zatrzymane na, odpowiednio instrukcjach *semB.acquire()* i *semC.acquire()* – zabrakło pozwoleń na wyświetlanie liter. Natomiast semafor *semA* miał wartość 1 – oznacza to, że jedno pozwolenie na wypisanie litery A nie zostało wykorzystane.

Proces *printA*() do wypisania pięciu liter A potrzebował pięciu pozwoleń. Dwa zostały przydzielone przez ustawienie wartości początkowej semafora *semA* – instrukcja *semA* = *Semaphore(2)*. W wynikowym

ciągu liter widzimy, że wypisane zostały dwie litery *C*. Po każdym wypisaniu litery *C* zostają wydane dwa pozwolenia na wypisanie litery *A*. W sumie, w naszym programie wydanych zostało sześć pozwoleń na wypisanie litery *A*. Ponieważ pętla w procesie *printA()* wykona się pięć razy, zostanie wykorzystanych tylko pięć pozwoleń z sześciu – jedno pozostanie zatem nie wykorzystane.

Dlaczego zabrakło pozwoleń na wypisywanie litery *B*? Wypisanie pojedynczej litery *B* wymaga dwóch pozwoleń (dwa razy wykonywana jest instrukcja *semB.acquire*()). Pozwolenia te wydawane są pojedynczo, w każdej iteracji w procesie *printA*(). Ponieważ iteracji tych jest tylko pięć, to pierwsze cztery pozwolenia, pochodzące z nich, zastaną wykorzystane na wypisanie dwóch liter *B*, natomiast ostatnie pozwoli procesowi *printB*() przejść przez pierwsze wywołanie metody *acquire*() na semaforze *semB*. Drugie wywołanie tej metody spowoduje zatrzymanie procesu *printB*(). Drogi Czytelniku, możesz to sprawdzić w prosty sposób, wstawiając dodatkowe polecenie wydruku między wywołaniami metody *acquire*(), tak jak jak to pokazano poniżej.

```
def printB():
    global COUNTER
    for i in range(COUNTER):
        semB.acquire()
        print('X ', end="")
        semB.acquire()
        print('B ', end="")
        semC.release()
```

Jeśli chodzi o proces *printC*(), to jego zatrzymanie jest konsekwencją zatrzymania procesu *printB*(), który wydaje pozwolenia na wypisanie litery *C*. Drogi Czytelniku, spróbuj zmienić wartość zmiennej *COUNTER* na sześć, sprawdź wyniki i przeanalizuj.

Z analizy zachowania procesów wynika, że proces *printA()* kończy swoje działanie zbyt wcześnie tzn. brakuje nam kolejnych jego iteracji, żeby mogły pojawić się dodatkowe pozwolenia na wypisanie litery *B*, a w konsekwencji również litery *C*. Jeżeli popatrzymy na ciąg, który chcemy uzyskać widzimy, że na każde dwie litery *A* powinna pojawić się para liter *B* i *C* – w ciągu liczba liter *A* jest dwa razy większa niż par liter *B C*. Rozwiązaniem naszego problemu blokady[3] jest zatem dwukrotne zwiększenie liczby iteracji w procesie *printA*(). Możemy to uzyskać modyfikując kod tego procesu, jak to pokazano poniżej.

```
def printA():
    global COUNTER
        for i in range(COUNTER*2): #parametr funkcji range przyjmuje wartość COUNTER*2
        semA.acquire()
        print('A ', end="")
        semB.release()
```

Po modyfikacji uzyskamy poniższy rezultat uruchomienia programu.

```
A A B C A A B C A A B C A A B C A A B C

threadA has ended
threadB has ended
```

[3]Blokowanie się procesów działających równolegle w systemie operacyjnym jest tematem niezwykle szerokim, więcej informacji na ten temat można znaleźć w [2].

```
threadC has ended
semA value is 2
semB value is 0
semC value is 0

All done

Process finished with exit code 0
```

Zwróćmy uwagę, że w wypisanym ciągu zadana sekwencja $A\ A\ B\ C$ powtarza się pięć razy, a wszystkie procesy zakończyły swoją pracę. Jeżeli chodzi o wartości semaforów, to jedynie semafor *semA* ma wartość różną od zero – posiada dwa niewykorzystane pozwolenia do wypisania litery A udzielone przez proces *printC*() po wypisaniu ostatniej litery C. Na bazie przedstawionego rozwiązania, spróbuj Drogi Czytelniku, zrealizować zadanie 1.

3. Synchronizacja procesów z koordynatorem

Rozważmy teraz nieco inny sposób rozwiązania naszego zadania. Do tej pory traktowaliśmy nasze procesy jako równorzędne – udzielały sobie nawzajem pozwoleń na wypisywanie liter, przekazując dostęp do konsoli (można to porównać do biegu sztafetowego z przekazywaniem pałeczki). Innym rozwiązaniem jest wyznaczenie jednego z procesów na koordynatora (kierownika) całej operacji. Rolą koordynatora będzie wydawanie pozwoleń (zleceń) na wypisywanie liter pozostałym procesom. Procesy te, po wykonaniu zadania (wydruku pojedynczej litery), będą informowały koordynatora o realizacji zlecenia. Na początek wybierzemy proces koordynatora – w naszym przypadku najlepiej sprawdzi się proces, który w zadanej sekwencji wypisuje pojedynczą literę. Mamy dwóch kandydatów. Wybierzemy proces *printB*(). Jego zadaniem będzie w każdej iteracji pętli, przed wypisaniem litery B, udzielenie dwóch pozwoleń na wypisanie litery A procesowi *printA*(), czyli wywołanie procedury V (*release*()) na odpowiednim semaforze. Następnie proces *printB()* będzie oczekiwać od procesu *printA*() potwierdzenia realizacji zlecenia. Po jego otrzymaniu wypisze swoją literę B i zleci procesowi *printC*() wypisanie litery C. Proces *printB*(), zanim przejdzie do kolejnej iteracji, poczeka na potwierdzenie od procesu *printC*(), że ten wykonał swoje zadanie.

Procesy *printA*() oraz *printC*() będą miały prostą, analogiczną strukturę. Każdy z nich przed wypisaniem litery zaczeka na odpowiednie pozwolenie wydane przez *printB*() wywołując procedurę P (*acquire*()) na odpowiednim semaforze. Po wypisaniu litery, oba procesy poinformują o tym fakcie proces *printB*() wywołując procedurę V (*release*()) na właściwym semaforze. Semafory wykorzystamy w następujący sposób:

- semafor *semA* posłuży koordynatorowi (procesowi *printB*()) do wydawania zleceń na wypisanie litery A – metoda *release*(), a procesowi *printA*() do przyjęcia zlecenia – metoda *acquire*();
- semafor *semC* posłuży koordynatorowi do wydawania zezwoleń na wypisanie litery C – metoda *release*(), a procesowi *printC*() do przyjęcia zlecenia – metoda *acquire*();
- semafor *semB* posłuży procesom *printA*() i *printC*() do informowania koordynatora o realizacji zlecenia – metoda *release*(), a koordynatorowi do przyjęcia potwierdzenia realizacji zlecenia – metoda *acquire*().

Kod naszych procesów, po wyznaczeniu procesu *printB*() na koordynatora, został przedstawiony poniżej. Bardzo istotne jest to, że wszystkie semafory mają wartości początkowe równe zero, nawet semafor *semA*. Proces *printB*() przejmuje kontrolę nad wydawaniem zleceń/pozwoleń na wypisywanie liter i jest odpowiedzialny za rozpoczęcie wypisywania sekwencji. Zwróćmy uwagę, że proces *printA*() nadal ma dwa razy większą liczbę iteracji niż pozostałe procesy. Czy jest to konieczne? Drogi Czytelniku, przekonaj się sam.

```
from threading import Thread, Semaphore

semA = Semaphore(0)
semB = Semaphore(0)
semC = Semaphore(0)
COUNTER = 5
```

```
   def printA():
       global COUNTER
           for i in range(COUNTER*2):
           semA.acquire()
           print('A ', end="")
           semB.release()
```

```
def printC():
    global COUNTER
    for i in range(COUNTER):
        semC.acquire()
        print('C ', end="")
        semB.release()
```

```
def printB():
    global COUNTER
    for i in range(COUNTER):
        semA.release()      #wydaj zlecenie wypisania jednej litery A
        semB.acquire()      #poczekaj na potwierdzenie wypisania
        semA.release()      #wydaj zlecenie wypisania jednej litery A
        semB.acquire()      #poczekaj na potwierdzenie wypisania
        print('B ', end="") #wypisz literę B
        semC.release()      #wydaj zlecenie wypisania jednej litery C
        semB.acquire()      #poczekaj na potwierdzenie wypisania
```

Procesy *printA*() oraz *printC*() używają semaforów w ten sam sposób – przed każdym wypisaniem swojej litery oczekują na odpowiednie zlecenie, a każdym po wypisaniu informują o tym proces koordynatora poleceniem *semB.release*(). Działanie procesu koordynatora przedstawiono w postaci komentarzy w powyższym kodzie. Rezultat uruchomienia naszych procesów, w zasadzie nie różni się od uruchomienia kodu bez koordynatora[4]. Jest jednak pewna drobna, ale istotna różnica – wartość semafora *semA* po zakończeniu programu wynosi zero. Nie zostały więc dwa, niewykorzystane pozwolenia na wypisanie litery *A*. Drogi Czytelniku, w ramach samodzielnych ćwiczeń spróbuj wykonać zadanie 2.

W naszym prostym przykładzie synchronizacja z koordynatorem nie była konieczna, ale możliwa. Są jednak przypadki, w których synchronizacja z koordynatorem jest najlepszym wyjściem lub takie, gdzie jest wręcz niezbędna (co pokażemy w kolejnym rozdziale).

Rozważmy ciąg $A\ A\ A\ B\ B\ C\ A\ A\ A\ B\ B\ C\ \ldots$. Widzimy, że na jedną literę C przypadają trzy litery A i dwie litery B. Zastosujemy rozwiązanie z koordynatorem – najbardziej odpowiednim kandydatem jest tu proces *printC*(). W związku z tym, semafor *semC* wykorzystamy do potwierdzania wykonania

[4]Cały czas stosujemy ten sam kod uruchamiający, wyświetlający dodatkowe informacje o procesach i semaforach.

zadań, zlecanych pozostałym procesom przez koordynatora. Jeżeli chcemy pozostać przy pętlach *for* w naszych procesach, musimy odpowiednio dostosować ich liczniki – to będzie Twoje zadanie Drogi Czytelniku. Kod procesów, z komentarzami zamieszczono poniżej, parametr *???* funkcji *range*() należy zastąpić odpowiednim wyrażeniem.

```
def printA():
    global COUNTER
    for i in range(???):
        semA.acquire()
        print('A ', end="")
        semC.release()
```

```
def printB():
    global COUNTER
    for i in range(???):
        semB.acquire()
        print('B ', end="")
        semC.release()
```

```
def printC():
    global COUNTER
    for i in range(???):
        semA.release()       #wydaj zlecenie wypisania jednej litery A
        semC.acquire()       #poczekaj na potwierdzenie wypisania
        semA.release()       #wydaj zlecenie wypisania jednej litery A
        semC.acquire()       #poczekaj na potwierdzenie wypisania
        semA.release()       #wydaj zlecenie wypisania jednej litery A
        semC.acquire()       #poczekaj na potwierdzenie wypisania
        semB.release()       #wydaj zlecenie wypisania jednej litery B
        semC.acquire()       #poczekaj na potwierdzenie wypisania
        semB.release()       #wydaj zlecenie wypisania jednej litery B
        semC.acquire()       #poczekaj na potwierdzenie wypisania
        print('C ', end="")  #wypisz literę C
```

Na bazie powyższego kodu, po jego uruchomieniu i przetestowaniu, spróbuj Drogi Czytelniku zrealizować zadanie 3.

4. Synchronizacja procesów z zewnętrznym koordynatorem

W poprzednich przykładach jeden z procesów wypisujących litery brał na siebie, dodatkowo rolę koordynatora. Rozważmy teraz nieco inny ciąg: $A\ B\ C\ C\ B\ A\ A\ B\ C\ C\ B\ A\ A\ B\ C\ C\ B\ A\ \ldots$. Powtarza się tu sekwencja $A\ B\ C\ C\ B\ A$, w której każda z liter występuje dwa razy, nie mamy więc dobrego kandydata na koordynatora. Poza tym, zauważmy, że w ciągu po literze B występują dwie litery C lub dwie litery A, zaś naszym zadaniem jest synchronizacja procesów jedynie przy pomocy semaforów, a nie dodatkowych instrukcji warunkowych w kodzie procesu wypisującego litery. Ponieważ żaden z procesów nie nadaje się na koordynatora, musimy powołać do życia czwarty proces (nazwijmy go *manager*()), którego jedynym zadaniem będzie zarządzanie procesami *printA*(), *printB*() i *printC*() w celu uzyskania właściwej sekwencji.

Zaczniemy od zdefiniowania niezbędnych semaforów. tym razem będziemy potrzebować czterech semaforów, zdefiniowanych w kodzie pokazanym poniżej.

```
semA = Semaphore(0)  #wypisywanie litery A
semB = Semaphore(0)  #wypisywanie litery B
semC = Semaphore(0)  #wypisywanie litery C
semM = Semaphore(0)  #informowanie kierownika o wykonanym zleceniu
```

Zdefiniujmy teraz proces koordynatora. Jak widać w poniższym kodzie, koordynator pracuje w pętli, w ramach której, w pojedynczej iteracji zleca wyświetlenie sekwencji *A B C C B A*.

```
def manager():
    global COUNTER
    for i in range(COUNTER):
        semA.release()   #wydaj zlecenie wypisania jednej litery A
        semM.acquire()   #poczekaj na potwierdzenie wypisania
        semB.release()   #wydaj zlecenie wypisania jednej litery B
        semM.acquire()   #poczekaj na potwierdzenie wypisania
        semC.release()   #wydaj zlecenie wypisania jednej litery C
        semM.acquire()   #poczekaj na potwierdzenie wypisania
        semC.release()   #wydaj zlecenie wypisania jednej litery C
        semM.acquire()   #poczekaj na potwierdzenie wypisania
        semB.release()   #wydaj zlecenie wypisania jednej litery B
        semM.acquire()   #poczekaj na potwierdzenie wypisania
        semA.release()   #wydaj zlecenie wypisania jednej litery A
        semM.acquire()   #poczekaj na potwierdzenie wypisania
```

Poniżej przedstawiono kod wszystkich trzech procesów odpowiedzialnych za wyświetlanie liter na konsoli. Zwróćmy uwagę, że znacznie się on uprościł. Każde wypisanie litery jest poprzedzone sprawdzeniem czy wydano na to pozwolenie (zlecenie). Po każdym wypisaniu jest ono potwierdzane za pomocą semafora *semM*.

```
   def printA():
       global COUNTER
       for i in range(COUNTER*2):
           semA.acquire()
           print('A ', end="")
           semM.release()
```

```
def printB():
    global COUNTER
        for i in range(COUNTER*2):
        semB.acquire()
        print('B ', end="")
        semM.release()
```

```
def printC():
    global COUNTER
    for i in range(COUNTER*2):
        semC.acquire()
        print('C ', end="")
        semM.release()
```

W tym przypadku należy zwrócić uwagę na liczbę iteracji pętli w procesach – jest ona dwa razy większa od liczby iteracji pętli w koordynatorze – wynika to faktu, że w ramach jednej iteracji tej pętli koordynator wydaje pozwolenia na wypisanie dwóch liter *A* i oczekuje potwierdzenia ich wypisania. Tak

samo jest w przypadku pozostałych liter B i C. Nie możemy więc dopuścić do sytuacji, w której proces wypisujący daną literę zakończy się zbyt wcześnie i koordynator wyda zlecenie wyświetlenia litery, które nie zostanie wykorzystane (proces wyświetlający wykonał swoje iteracje), a tym samym koordynator nie doczeka się na potwierdzenie wypisania i zostanie zablokowany.

Zanim będzie możliwe przetestowanie naszego przykładu, musimy jeszcze uzupełnić kod uruchamiający nasze procesy uwzględniając nowy proces koordynatora. Kod ten pokazano poniżej.

```
threads = []
threads.append(Thread(target=printA, name="threadA", daemon=True))
threads.append(Thread(target=printB, name="threadB", daemon=True))
threads.append(Thread(target=printC, name="threadC", daemon=True))
threads.append(Thread(target=manager, name="threadM", daemon=True))

for thread in threads:
    thread.start()
for thread in threads:
    thread.join(timeout=1)
print("\n")
for thread in threads:
    if(thread.isAlive()):
        print(thread.getName()+" is alive")
    else:
        print(thread.getName() + " has ended")
print('semA value is {}'.format(semA._value))
print('semB value is {}'.format(semB._value))
print('semC value is {}'.format(semC._value))
print('semM value is {}'.format(semM._value))
print("\nAll done")
sys.exit()
```

Uruchomienie kodu programu wyświetli poprawną sekwencję znaków, co pokazano poniżej.

```
A B C C B A A B C C B A A B C C B A A B C C B A A B C C B A

threadA has ended
threadB has ended
threadC has ended
threadM has ended
semA value is 0
semB value is 0
semC value is 0
semM value is 0

All done
```

Zwróćmy szczególną uwagę na informacje o stanie procesów i semaforów, wyświetlone po wypisaniu ciągu liter – wszystkie procesy się zakończyły poprawnie, a wartości wszystkich semaforów są równe zero.

Drogi Czytelniku, w ramach analizy zadania wypróbuj jak zadziała nasz kod w sytuacji kiedy kody procesów wypisujących litery będą miały liczbę iteracji w pętli *for* ustawioną na wartość *COUNTER* (*for i in range*(*COUNTER*)). Wiedząc jak wykorzystać proces koordynatora, możesz poćwiczyć uzyskiwanie różnych sekwencji (również uwzględniających dodatkowe litery), których propozycje znajdziesz w zadaniu 4.

5. Podsumowanie

W artykule pokazano jak ograniczenie liczby iteracji w pętlach, w procesach pracujących równolegle może wpłynąć negatywnie na ich działanie i jak sobie z tym poradzić. Omówiono również sposoby wykorzystania procesu koordynatora do zarządzania działaniem innych procesów. Proste przykłady, które Czytelnik może uruchomić samodzielnie, pozwalają na lepsze zrozumienie omawianych zagadnień oraz zachęcają do eksperymentowania, które pomaga zrozumieć zjawiska z jakimi możemy mieć do czynienia przy synchronizacji procesów. Zadania do samodzielnego wykonania stanowią dobre ćwiczenie nabytych umiejętności. Powinny również zachęcić do realizacji własnych pomysłów na realizowane sekwencje.

6. Zadania do samodzielnego wykonania

Zadanie 1. Dla zaprezentowanego w treści artykułu przykładu, proszę spróbować uzyskać następujące sekwencje liter:

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

pamiętając, że używamy tylko mechanizmu semaforów do synchronizacji procesów i nie modyfikujemy funkcji *print*. Proszę również zaobserwować, jak zmiany wartości początkowej zmiennej *COUNTER* wypłyną na wyświetlane wyniki.

Zadanie 2. Dla zaprezentowanego w treści artykułu przykładu, spróbować zmodyfikować kod tak, żeby proces *printC*() przejął rolę koordynatora. Następnie proszę spróbować uzyskać następujące sekwencje, pamiętając o wybraniu właściwego procesu na koordynatora:

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

Proszę zwróć uwagę na liczniki pętli w procesach.

Zadanie 3. Na bazie kodu, zaprezentowanego w treści artykułu, proszę spróbować uzyskać następujące sekwencje liter:

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

pamiętając o wybraniu właściwego procesu na koordynatora oraz o doborze odpowiednich liczników pętli *for*.

Zadanie 4. Na bazie zaprezentowanego w treści artykułu kodu wykorzystującego zewnętrzny proces koordynatora, proszę spróbować uzyskać następujące sekwencje liter:

- *A B A B A C C A A B A B A C C A*...,
- *A B B C C C B B A A B B C C C B B A* ...,

pamiętając o doborze odpowiednich liczników pętli *for*. Definiując dodatkowy proces wypisujący literę *D*, proszę spróbować uzyskać następujące sekwencje:

- *A B C D D C B A*...,
- *A B B A C C A D D A B B A C C A D D*....

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.[5]
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. I*, MINUT 2020 (2), s. 41-49.
5. *The Python Standard Library, Synchronization Primitives*, Python Software Foundation.

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