MINUT 2019 (1), s. 17–23
ISSN 2719-3063

Ewa PŁUCIENNIK

Wydział Automatyki, Elektroniki i Informatyki, Politechnika Śląska

ul. Akademicka 16, 44-100 Gliwice

Semafory jako mechanizm synchronizacji procesów w systemie operacyjnym – wprowadzenie

Streszczenie. Artykuł przedstawia krótkie wprowadzenie do tematyki synchronizacji procesów, pracujących równolegle w systemie operacyjnym komputera, z wykorzystaniem mechanizmu semaforów. Po ogólnym omówieniu zasad funkcjonowania semaforów, ich działanie przedstawiono na prostym przykładzie praktycznym, realizowanym z wykorzystaniem języka Python.

1. Wstęp

Każdy z nas posługując się urządzeniem typu komputer czy smartfon może z niego korzystać dzięki systemowi operacyjnemu. To on odpowiedzialny jest m. in. za umożliwienie różnym procesom dostępu do zasobów takich jak procesor, pamięć czy drukarka oraz zapewnienie bezkonfliktowej współpracy. Trudno dziś wyobrazić sobie jednozadaniowy system operacyjny czyli taki, który realizuje tylko jeden proces w danym momencie. Tak więc nasze systemy operacyjne są wielozadaniowe co oznacza, że w danej chwili pracuje w nich wiele procesów, a co z tym idzie procesy te konkurują ze sobą o dostęp do zasobów, ale również współpracują ze sobą, żeby zrealizować jakieś działanie.

W artykule przyjrzymy się jak można wpłynąć na tę współpracę, synchronizując równolegle działające procesy z wykorzystaniem mechanizmu nazywanego semaforem. Analogia do semafora kolejowego nasuwa się sama. Podniesienie semafora informuje maszynistę, że wjazd pociągu na tor jest możliwy i bezpieczny, opuszczenie jest sygnałem, że wjazd na tor, z jakiejś przyczyny, jest w danym momencie niemożliwy – może trzeba przepuścić pociąg o wyższym priorytecie czyli np. pociąg lokalny ustępuje miejsca pociągowi międzynarodowemu. Tak działający mechanizm, zaimplementowany w systemie operacyjnym, pozwala sterować wykonaniem równolegle działających procesów dając możliwość zatrzymania danego procesu tak, żeby np. poczekał aż inny proces wykona jakieś działania (obliczenia). Semafor komputerowy składa się z kilku elementów. Pierwszym z nich jest tzw. zmienna semaforowa, która jest liczbą całkowitą, a jej wartości początkowa jest nieujemna. Drugim elementem semafora jest stowarzyszona z nim kolejka, w której umieszczane są procesy, które zgłosiły potrzebę "przejścia" przez semafor, który okazał się zamknięty – można to porównać do pociągu stojącego przed opuszczonym semaforem kolejowym[1].

Autor korespondencyjny: E. Płuciennik (Ewa.Płuciennik@polsl.pl).
Data wpłynięcia: 11.08.2019.

[1]w przypadku semafora kolejowego kolejka może mieć tylko jeden element, w systemie komputerowym może być dłuższa

Semafor komputerowy, którym możemy się posłużyć do zatrzymania procesu może być tzw. semaforem binarnym czyli semaforem, który ma dwa stany: otwarty (podniesiony) lub zamknięty (opuszczony). Zmienna semaforowa takiego semafora przyjmuje wartość 1 jeśli semafor jest otwarty, a wartość 0 jeśli semafor jest zamknięty. Semafor binarny jest szczególnym przypadkiem semafora ogólnego, w którym zmienna semaforowa może przyjmować dowolne wartości całkowite (interpretacja tych wartości zostanie wyjaśniona w dalszej części artykułu, przy omawianiu przykładu praktycznego). Definiując zmienną semaforową musimy nadać jej wartość początkową. Jest to operacja, która jest realizowana poza procesami, które będą korzystać z semafora. Załóżmy, że zdefiniujemy zmienną semaforową S i nadamy jej wartość początkową 0. Oznacza to, że semafor S jest zamknięty. Ze zmienną semaforową stowarzyszone są dwie operacje, które mogą wykonać na niej procesy chcące skorzystać z semafora:

- próba przejścia przez semafor tzw. procedura *P(S)* – polega na dekrementacji[2] 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[3] zmiennej semaforowej i sprawdzeniu czy jej wartość jest większa od zera, jeśli nie oznacza to, że jakiś proces czeka przed semaforem – wtedy jest on przepuszczany przez semafor.

Korzystanie z semaforów ma sens tylko wtedy, kiedy mamy do czynienia z procesami działającymi równolegle. Załóżmy, że mamy dwa takie procesy, które dokonują pewnych obliczeń. Zapiszemy ich treść ogólnie, w postaci pseudokodu.

```
Proces 1{                                  Proces 2{
oblicz A;                                  oblicz C;
oblicz B; }                                oblicz D korzystając z wartości B; }
```

Co się stanie, jeżeli proces 1 nie zdąży obliczyć B zanim proces 2 będzie chciał obliczyć D? Oczywiście nasze procesy są opisane zbyt ogólnie, żeby konkretnie odpowiedzieć na to pytanie, ale możemy przypuszczać, że proces drugi może mieć wtedy problem – być może nie zakończy się poprawnie, może źle obliczy wartość D. Żeby mieć pewność, że proces 2 nie zacznie obliczać D zanim proces 1 nie obliczy B możemy zastosować semafor, przez który będzie musiał "przejść" proces 2, wykonując procedurę *P(S)*, zanim przystąpi do obliczeń D. Semafor na początku (przy inicjalizacji, która odbędzie się zanim oba procesy wystartują) zostanie zamknięty. Za otwarcie semafora, czyli wykonanie procedury *V(S)*, odpowiedzialny będzie proces 1, a uczyni to dopiero po wyznaczeniu wartości B. Zapiszmy to w pseudokodzie zakładając, że mamy utworzony semafor S i nadaliśmy mu wartość początkową 0.

```
Proces 1{                                  Proces 2{
oblicz A;                                  oblicz C;
V(S); //otwórz semafor                     P(S); //czekaj na otwarcie semafora
oblicz B; }                                oblicz D korzystając z wartości B; }
```

Nasz ogólny przykład jest bardzo prosty i zapisany w pseudokodzie. Spróbujmy przyjrzeć się działaniu semaforów w praktyce i na nieco bardziej skomplikowanym przykładzie. Wykorzystamy w tym celu język Python[4]. Python jest językiem cieszącym się coraz większą popularnością, szczególnie jeśli chodzi

[2] zmniejszeniu o 1

[3] zwiększeniu o 1

[4] https://www.python.org/ - tu Czytelnik znajdzie nie tylko pełną dokumentację Pythona, ale także samouczki

o naukę programowania dla początkujących [1]. Do uruchomienia przykładów prezentowanych w artykule przydatne (ale nie niezbędne) będzie narzędzie Pycharm[5].

2. Procesy działające równolegle

Zdefiniujemy trzy procesy, które operować będą na wspólnych zmiennych globalnych. W języku Python, żeby móc posłużyć się semaforami dla równolegle działających procesów (zwanych również wątkami[6]) musimy zaimportować odpowiednie elementy (*Semaphore* oraz *Thread*) z pakietu *threading*. Wykorzystamy również pakiet *time*, żeby móc zasymulować pracę wątków w dłuższej perspektywie czasowej (w naszym przypadku będzie to około 0.5 sekundy). Następnie zdefiniujemy trzy zmienne globalne A, B i C oraz nadamy im wartości początkowe wynoszące zero. W języku Python nie musimy deklarować typu tych zmiennych, zostanie on określony na podstawie przypisanych im wartości (jest to tzw. typowanie dynamiczne).

```
from threading import Semaphore, Thread # import niezbędnych elementów
import time

A = 0 # deklaracja zmiennych
B = 0
C = 0
```

Następnie zdefiniujemy trzy procesy, które po pewnych obliczeniach (zasymulowanych instrukcją *time.sleep()* z taką samą wartością dla każdego z procesów – zakładamy, że obliczenia wykonywane przez procesy są podobnej klasy) nadadzą zmiennym globalnym właściwe wartości, przy czym proces 1 wyznaczy wartość zmiennej A, proces 2 zmiennej B, a proces 3 zmiennej C. Oprócz tego procesy będą, w dalszej kolejności, wykorzystywać wartości zmiennych do kolejnych obliczeń. I tak, proces 1 musi wyznaczyć sumę A i B, proces 2 sumę B i C, a proces 3 sumę A, B i C.

Szczególnie istotna z naszego punktu widzenia jest deklaracja *global A*, która oznacza, że w procesie nie deklarujemy nowej zmiennej A, ale korzystamy ze zmiennej globalnej. Żeby przetestować prezentowany kod, należy wpisać go w jednym pliku z rozszerzeniem *py* i uruchomić np. przy pomocy narzędzia Pycharm.

```
def process_1():
  global A
  time.sleep(0.5)
  A = 10
  print(f"P1 sum: {A+B}")
```

```
def process_2():
  global B
  time.sleep(0.5)
  B = 20
  print(f"P2 sum: {B+C}")
```

```
def process_3():
  global C
  time.sleep(0.5)
  C = 30
  print(f"P3 sum: {A+B+C}")
```

Mając tak zdefiniowane procesy musimy je jeszcze uruchomić tak, żeby działały jako równoległe wątki. W tym celu stworzymy tablicę o nazwie *threads*, a następnie dodamy do niej wszystkie zdefiniowane procesy metodą *append()*. Iterując, w pętli *for*, po tej tablicy uruchamiany wszystkie wątki metodą *start()*. Kolejna pętla *for*, z wywołaniem metody *join()* dla każdego wątku, służy do tego żeby poczekać na zakończenie pracy wszystkich wątków. Potem w celach kontrolnych wyświetlimy sumę zmiennych globalnych A, B i C

[5] https://www.jetbrains.com/pycharm/

[6] proces uważa się za coś większego od wątku tzn. dany proces może być realizowany przez kilka wątków, w niniejszym artykule pojęcie wątek i proces jest stosowane zamiennie

```
threads = []            # deklarujemy tablicę
threads.append(Thread(target=process_1)) # dodajemy procesy jako wątki
threads.append(Thread(target=process_2))
threads.append(Thread(target=process_3))

for thread in threads:  # uruchamiamy wątki
    thread.start()
for thread in threads:  # czekamy na zakończenie wątków
    thread.join()
print("\nAll done")     # wątki zakończyły pracę
print(f"sum: {A + B + C}")
```

Postawmy następujące pytania: jakie wartości sum wypiszą poszczególne wątki, jaka wartość sumy pojawi się na końcu? Żeby na nie odpowiedzieć wystarczy oczywiście uruchomić nasz kod, ale żeby mieć pewność uruchomimy go kilkukrotnie. Wyniki trzykrotnego uruchomienia naszego programu są następujące.

```
P1 sum: 10              P3 sum: 30              P2 sum: 20
P3 sum: 40              P2 sum: 50              P3 sum: 50
P2 sum: 50              P1 sum: 30              P1 sum: 30

All done                All done                All done
sum: 60                 sum: 60                 sum: 60
```

Jakie wnioski możemy wyciągnąć z tych wyników. Na początek zauważmy, że suma wypisana na końcu zawsze wynosi 60. Dzieje się tak dlatego, że jej obliczenie wykonywane jest dopiero wtedy kiedy wszystkie wątki zakończą swoją pracę i nadadzą zmiennym A, B i C odpowiednio wartości 10, 20 i 30. Mamy pewność, że obliczenie sumy nastąpi dopiero po nadaniu składnikom sumy tych wartości.

Inaczej jest w przypadku sum wypisywanych przez poszczególne wątki. Zwróćmy uwagę, że wątek 3 wypisujący sumę wszystkich zmiennych też wypisał różne wartości. Stało się tak dlatego, że nasze wątki wykonują się równolegle i nie wiemy czy wątki 1 i 2 zdążą nadać wartości zmiennym A i B zanim wątek 3 wypisze sumę. Jedyne czego możemy być pewni to to, że wartość zmiennej C wyniesie 30 zanim wątek 3 wypisze sumę, ponieważ operacje w ramach wątku są wykonywane sekwencyjnie, czyli jedna po drugiej. Podobnie możemy być pewni, że w przypadku wątku 1 przed obliczeniem sumy $A + B$ wartość A wyniesie 10, a w przypadku wątku 2 przed obliczeniem sumy $B + C$ wartość B wyniesie 20. Kolejność wyświetlania sum z poszczególnych procesów mówi nam o tym, które wątki były szybsze w swoich obliczeniach.

3. Synchronizujemy procesy

Nasze wątki są w zasadzie nieprzewidywalne w swoich działaniach. Jest to sytuacja, która może rodzić duże problemy. Oczywiście można by stwierdzić, że w takim razie należy wszystkie obliczenia wykonać sekwencyjne i mieć pewność co do wyników. Wtedy jednak wydłużamy znacznie czas oczekiwania na rezultaty i rezygnujemy z wykorzystania możliwości równoległego wykonywania obliczeń. Lepiej postarać się zsynchronizować wątki, czyli powiedzieć im kiedy mają poczekać aż inny wątek dokona jakiś obliczeń, a kiedy mogą pracować swobodnie. Jak to powinno wyglądać w naszym przypadku:

- proces 1 musi poczekać, aż proces 2 obliczy wartość B, żeby wypisać sumę A i B;

- proces 2 musi poczekać, aż proces 3 obliczy wartość C, żeby wypisać sumę B i C;
- proces 3 musi poczekać, aż procesy 1 i 2 wyznaczą wartości A i B, żeby wypisać sumę A, B i C.

Zacznijmy od realizacji pierwszego punktu, czyli każemy procesowi 1 poczekać na wyznaczenie wartości B przez proces 2. Posłużymy się w tym celu semaforem, który zdefiniujemy zaraz po deklaracji zmiennych globalnych, nadając mu wartość początkową 0 (ustawienie wartości początkowej odbywa się poza procesami):

```
semB = Semaphore(0)
```

Nasz semafor *semB* będzie odpowiedzialny za kontrolę przetwarzania zmiennej globalnej B. Wartość początkowa semafora (zmiennej semaforowej) wynosi 0, co oznacza, że semafor jest zamknięty czyli informuje nas o tym, że wartość B nie została jeszcze wyznaczona. Użyjmy naszego semafora, żeby proces 1 wiedział kiedy musi poczekać, aż proces 2 obliczy B. Z kolei proces 2, korzystając z tego samego semafora poinformuje proces 1, że wyznaczył wartość zmiennej B:

```
def process_1():
  global A
  time.sleep(0.5)
  A = 10
  semB.acquire() # P(semB) czekam
  print(f"P1 sum: {A + B}")
```

```
def process_2():
  global B
  time.sleep(0.5)
  B = 20
  semB.release() # V(semB) podnoszę semafor
  print(f"P2 sum: {B + C}")
```

W Pythonie (i nie tylko w nim) odpowiednikiem procedury P jest metoda *acquire()*, a procedury V metoda *release()*, wykonywane na rzecz obiektu *semB* reprezentującego nasz semafor [3]. Efekt jaki uzyskaliśmy jest taki, że oba procesy zaczynają działać równolegle, natomiast proces 1 zatrzymuje się na instrukcji *semB.acquire()* (semafor jest opuszczony), aż do momentu kiedy proces 2 wykona operację *semB.release()* podnosząc semafor. Oto wyniki trzykrotnego uruchomienia naszego programu w nowej wersji:

```
P3 sum: 30
P2 sum: 50
P1 sum: 30

All done
sum: 60
```

```
P2 sum: 20
P1 sum: 30
P3 sum: 60

All done
sum: 60
```

```
P2 sum: 20
P1 sum: 30
P3 sum: 60

All done
sum: 60
```

Możemy zauważyć, że proces 1 zawsze wypisał tę samą wartość sumy i będzie tak za każdym razem, nawet jeśli zmienimy czasy w instrukcji *time.sleep()* w poszczególnych wątkach. Dzieje się tak, ponieważ dokonaliśmy częściowej synchronizacji wątków. Częściowej dlatego, że nie uwzględniliśmy jeszcze pozostałych niuansów obliczeń, jakie wykonują nasze wątki (widać to w wartościach sum wypisywanych przez procesy 2 i 3). Dokończmy zatem synchronizację, definiując kolejne dwa semafory *semA* i *semC* odpowiedzialne za synchronizację obliczeń wartości, odpowiednio zmiennej B i C. Na początku wszystkie semafory są zamknięte, ponieważ wartości zmiennych jeszcze nie zostały wyznaczone.

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

Proces 1, po wyznaczeniu A, informuje o tym podniesieniem semafora *semA*, a następnie próbuje przejść przez semafor pilnujący obliczenia zmiennej B. Proces 2, po wyznaczeniu wartości B, informuje o tym podnosząc semafor *semB* i próbuje przejść przez semafor pilnujący obliczenia zmiennej C. Z kolei proces 3, po wyznaczeniu wartości C, informuje o tym podnosząc semafor *semC*, a następnie próbuje przejść, kolejno przez semafor *semB* i *semA* ponieważ musi mieć pewność, że zmienne A i B zostały już wyznaczone.

```
def process_1():
  global A
  time.sleep(0.5)
  A = 10
  semA.release()
  semB.acquire()
  print(f"P1 sum: {A+B}")
```

```
def process_2():
  global B
  time.sleep(0.5)
  B = 20
  semB.release()
  semC.acquire()
  print(f"P2 sum: {B+C}")
```

```
def process_3():
  global C
  time.sleep(0.5)
  C = 30
  semC.release()
  semB.acquire()
  semA.acquire()
  print(f"P3 sum: {A+B+C}")
```

Uruchommy nasz kod... efekt nie jest do końca satysfakcjonujący:

```
P2 sum: 50
P1 sum: 30

Process finished with exit code -1
```

Program musiał być ręcznie przerwany, ponieważ nie można było doczekać się na wyniki z procesu 3. Kolejne uruchomienie kodu dało następujący wynik:

```
P2 sum: 50
P3 sum: 60

Process finished with exit code -1
```

Tym razem proces 1 nie był w stanie dokończyć swojego działania. Dokładną analizę, dlaczego w tym przypadku, pomimo uruchamiania takiego samego kodu, wyniki są różne, pozostawiam Czytelnikowi w ramach ćwiczeń praktycznych (można też zmieniać wartość w instrukcji *time.sleep()* w poszczególnych procesach, symulując większe lub mniejsze opóźnienia w pracy procesów).

Zarówno w pierwszym, jak i drugim przypadku problemem jest to, że proces 2 informując o tym, że obliczył wartość B zwiększa wartość semafora *semB* o jeden (z 0 na 1), co można porównać do wydania jednej przepustki do przejścia przez semafor. Przejść przez semafor chcą dwa procesy 1 i 3. Ten, który pierwszy to zrobi wykorzysta przepustkę (zmniejszy wartość semafora *semA* z 1 na 0) i sprawi, że semafor znowu się zamknie.

Jak możemy to rozwiązać? Jeśli chcemy pozostać przy semaforach binarnych możemy zdefiniować dwa semafory, dzięki którym proces 2 poinformuje osobno proces 1 i 3 o zakończeniu obliczeń zmiennej A (Drogi Czytelniku, w ramach ćwiczeń, spróbuj zaimplementować takie rozwiązanie). Wykorzystanie semaforów binarnych w naszym prostym przykładzie wymaga w tym momencie zdefiniowania czterech semaforów. Gdyby okazało się, proces 2 musi poinformować więcej innych procesów o tym, że wyznaczył wartość B, to dla każdego z nich musiałby użyć osobnego semafora, co spowolniło by proces 2.

Znacznie lepszym rozwiązaniem będzie wykorzystanie, zamiast semafora binarnego, semafora ogólnego czyli takiego, którego zmienna semaforowa może również przyjmować wartości większe od jeden.

Co oznacza wartość zmiennej semaforowej wynoszącej więcej niż jeden np. trzy? Oznacza to, że w danym momencie semafor może przepuścić trzy procesy. Działanie semafora ogólnego możemy porównać do biletera wpuszczającego zwiedzających na wystawę sztuki. Załóżmy, że nasza wystawa odbywa się w pomieszczeniu, w którym mieści się 30 osób, ale wystawa cieszy się sporym zainteresowaniem i chce ją odwiedzić znacznie więcej osób. Ustawiają się one w kolejce przed wejściem. Bileter (semafor S) ma na początku do dyspozycji 30 wejściówek. Każdy zwiedzający (proces) wchodzący na wystawę otrzymuje od biletera jedną wejściówkę (tym samym zmniejsza się pula dostępnych wejściówek - każdy chcący wejść na wystawę wykonuje procedurę *P(S)*). Kiedy okaże się, że pula wejściówek się wyczerpała, pozostali chętni do obejrzenia wystawy będą musieli czekać w kolejce. Osoby zwiedzające w końcu zdecydują się na wyjście z wystawy. Wychodząc zwracają bileterowi wejściówkę, którą od niego dostały (wykonują procedurę *V(S)*). Ponieważ w kolejce czekają następni chętni do zwiedzania, bileter przekazuje zwracane wejściówki kolejnym oczekującym osobom. Wartość zmiennej semaforowej waha się w tym przypadku od wartości początkowej wynoszącej 30, przez wartość 0 (oznaczającą, że wystawę zwiedza aktualnie 30 osób i nikt nie czeka w kolejce), po wartości ujemne, które oznaczają, że wystawa jest pełna zwiedzających i są osoby, które czekają w kolejce. Na przykład, wartość zmiennej semaforowej wynosząca -20 oznacza, że w kolejce czeka 20 osób.

W naszym przypadku, wykorzystanie semafora ogólnego oznacza, że proces drugi po obliczeniu wartości B powinien wydać dwie przepustki do semafora *semB* (dwa razy wykonując polecenie *semA.release()*), z których natychmiast będą mogły skorzystać procesy 1 i 3. Wynik działania tak zmodyfikowanego programu będzie zawsze taki sam, tzn. procesy 1, 2, 3 za każdym razem wypiszą wartości sum odpowiednio: 30, 50, 60. różnica może być jedynie w kolejności wypisania. Modyfikację kodu i testy pozostawiam do implementacji Czytelnikowi w ramach ćwiczeń praktycznych.

4. Podsumowanie

Artykuł miał na celu wstępne zapoznanie czytelnika z problemem synchronizacji procesów równoległych w systemie operacyjnym oraz zachęcenie do ćwiczeń praktycznych, które można przeprowadzić w zasadzie w dowolnym języku programowania, korzystając z oferowanych w nim mechanizmów obsługi semaforów. Semafory to bardzo prosty mechanizm, sprowadzający się właściwie do operacji inkrementacji i dekrementacji liczby całkowitej oraz możliwości wstrzymania działania procesu. Wykorzystując semafory możemy nie tylko synchronizować procesy, ale również kolejkować je w oczekiwaniu na dostępność zasobów, jak również umożliwić procesom komunikację, czyli przekazywanie sobie informacji. Oczywiście semafory nie są jedynym mechanizmem wykorzystywanym w przypadku pracy równoległej [2], ale zrozumienie zasady ich działania na pewno będzie bardzo przydatne dla osób rozpoczynających przygodę z programowaniem, również tym wielowątkowym.

Literatura

1. J.A. Briggs, *Python dla dzieci. Programowanie na wesoło.* PWN, Warszawa 2016.
2. A.B. Downey, *The Little Book of Semaphores.*[7] 2016.
3. *The Python Standard Library, Synchronization Primitives.* Python Software Foundation

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