Gliwice, 18.06.2018



**Obliczenia Równoległe**

Temat:

Problem ucztujących filozofów

Bartłomiej Krasoń

Informatyka, sem. 4, gr. 6

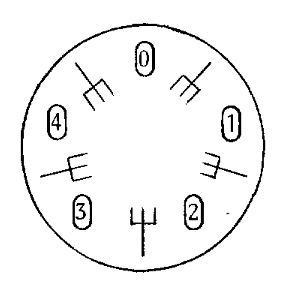
[bartkra814@student.polsl.pl](mailto:bartkra814@student.polsl.pl)

# Wstęp

*„Problem ucztujących filozofów”* jest klasycznym problemem natury programowania współbieżnego. Jest to teoretyczne wyjaśnienie zjawiska zakleszczenia i ograniczenia dostępu innym jednostką do zasobów współdzielonych. Co ciekawe, geneza sformułowania tego problemu jest związana z zadaniem egzaminacyjnym, wymyślonym przez samego **Edsgera Djikstrę**, wybitnego naukowca uważanego za pioniera informatyki, które dotyczyło sytuacji, w której 5 różnych komputerów, próbuje uzyskać dostęp do 5 współdzielonych napędów dyskowych. Następnie problem ten został przekształcony w tytułowy *„Problem ucztujących filozofów”* przez **sir Charlesa Hoare’a**, cenionego w świecie informatyki naukowca, m.in. za zaproponowany przez niego 1się nad nim największe autorytety dyscypliny naukowej, jaką jest informatyka.

## *Sformułowanie problemu:*

*Pięciu filozofów, ponumerowanych od 0 do 4 żyje w domu, w którym leży jeden stół, każdy filozof ma własne miejsce przy stole. Ich jedynym problemem – oprócz filozofii – jest to, że serwowanym daniem jest spaghetti, które musi być spożywane dwoma widelcami. Obok każdego talerza znajdują się dwa widelce, co nie stanowi problemu; jednakże w konsekwencji, dwóch sąsiadów nie może jeść jednocześnie.*

**

1

0

4

3

2

*Rys. 1 - Oryginalna ilustracja, prezentująca graficznie „Problem ucztujących filozofów” pochodząca z wykłady pt. „Hierarchial Ordering of Sequential Processes” prof. E.Djikstry.*

Sformułowane zagadnienie może wydawać się proste z punktu widzenia życia codziennego, jednakże nas interesuje rozwiązanie powyższego problemu w dziedzinie informatyki, który polega na synchronizacji procesów pracujących ze sobą współbieżnie na współdzielonych zasobach. Procesami w sformułowanym przez nas problemie jest piątką wspomnianych *filozofów*, natomiast zasobami są *widelce*. Zaś czynnościami (procedurami) które filozofowie (procesy) mogą wykonywać są naprzemiennie występujące, *myślenie* – nie wymagające żadnego zasobu oraz *jedzenie* – wymagające pary widelców.

Przedstawienie *„problemu ucztujących filozofów”* w postaci kodu programu, może przyjąć ogólny schemat:

**task** Filozofi *-- jeden z 5 filozofów*

**task body** Filozofi **is**

**begin**

**loop**

Myśl;

Jedz;

**end loop**;

**end** Filozofi;

# Rozwiązania

## Rozwiązanie naiwne z wykorzystaniem semaforów

**Semaforem** nazwiemy jednostkę, w postaci zmiennej chroniącej dostępu do danego zasobu w programie. W naszym przypadku każdy widelec będzie posiadał swój *semafor binarny*, tj. przyjmujący wartości:

**0** – gdy widelec jest obecnie w użyciu,

**1** – gdy widelec jest dostępny.

Na semaforach możliwe są do wykonania dwie następujące procedury:

**P** – dekrementuje wartość semafora, inaczej nazywana: wait // pobierz zasób

**V** – inkrementuje wartość semafora, inaczej nazywana: signal // zwolnij zasób

Przy realizacji zadania, przyjmiemy indeksowanie filozofów i widelców zgodne z zaprezentowanym na rysunku *Rys.1*. Na lewo od *i-tego* filozofa leży *i-ty* widelec, na prawo *i+1-wszy* mod 5.

Widelec: array(0…4) of natural := (1,1,1,1,1)

**task** Filozofi *-- jeden z 5 filozofów*

**task body** Filozofi **is**

**begin**

**loop**

Myśl;

P(Widelec(i)); *-- pobierz lewy widelec*

P(Widelec((i+1) mod 5)); *-- pobierz prawy widelec*

Jedz;

V(Widelec(i)); *-- zwolnij lewy widelec*

V(Widelec((i+1) mod 5)); *-- zwolnij prawy widelec*

**end loop**;

**end** Filozofi;

Taka próba rozwiązania problemu nie jest jednak dopuszczalna. Mimo iż spełnione zostaje założenie, że żadna para sąsiadów nie może jeść jednocześnie, powstaje jednak nowy problem, a mianowicie możliwość **zakleszczenia**. Sytuacja ta maiłaby miejscy, gdyby wszyscy filozofowie na raz wzięli po lewym widelcu. Wtedy każdy czekał by na prawy, co spowodowałoby nieoczekiwane zawieszenie programu – a raczej kolacji.

## Rozwiązanie z równoczesnym podnoszeniem widelców

Jednakowoż i ten problem ma kilka swoich rozwiązań. Jednym z nich, najbardziej naturalnym wydaje się być koncepcja z równoczesnym podnoszeniem widelców, tj. zakładamy, że operacja pobierania widelców lewego i prawego jest atomowa – nierozłącznie zawsze występuje razem.

Widelec: array(0…4) of natural := (1,1,1,1,1)

**task** Filozofi *-- jeden z 5 filozofów*

**task body** Filozofi **is**

**begin**

**loop**

Myśl;

P(Widelec(i), Widelec((i+1) mod 5));

*-- ^ pobierz lewy i prawy widelec*

Jedz;

V(Widelec(i), Widelec((i+1) mod 5));

*-- ^ zwolnij lewy i prawy widelec*

**end loop**;

**end** Filozofi;

Takie podejście eliminuje problem wystąpienia *zakleszczenia*, jednakże nadal nie jest ono w stu procentach idealne. Może dojść do tak zwanego **zagłodzenia**. Jest to sytuacja, w której pewnie filozof nie potrafi uzyskać dostępu do obu widelców na raz. Innymi słowy, zawsze co najmniej jeden z sąsiadów danego filozofa, jest w trakcie posiłku. Mimo iż mógłby on uzyskać dostęp do jednego z widelców, czeka aż drugi się zwolni, lecz w trakcie oczekiwania, może utracić dostęp znowu do pierwszego. *Zagłodzenie* jest powszechnym problemem natury programowania współbieżnego. Jest jednym z wielu problemów, z którym radzić sobie muszą projektanci systemów operacyjnych. Jednym ze sposobów radzenia sobie z *zagłodzeniem* jest ustalenie pewnego maksymalnego czasu oczekiwania procesu na dostęp do danego zasobu, po przekroczeniu którego następuje wywłaszczeni tego zasobu na rzecz oczekującego procesu.

## Rozwiązanie z pośrednim stanem

W tym rozwiązaniu każdemu filozofowi dodajemy zmienną *C[i]*, określającą jego stan. Zmienna przyjmuje następujące wartości:

C[i]=0 – filozof *i-ty* myśli,

C[i]=1 – filozof *i-ty* jest głodny // stan pośredni

C[i]=2 – filozof *i-ty* je,

Następnie rozważamy, jakie stany są niedopuszczalne. Jest to stan gdy pewien filozof oraz jego lewy sąsiad jedzą na raz, czyli:

Aby zapobiec takiej sytuacji, przyjmiemy, że i-ty filozof może przystąpić tylko w określonej, bezpiecznej sytuacji do jedzenia, a następuje ona gdy:

- filozof *i-ty* jest głodny

- filozof *i+1-wszy* nie je

- filozof *i-1-wszy* nie je

Wykrycie takiej sytuacji, zagwarantuje nam specjalnie zdefiniowana procedura *test*:

**procedure** test (K: in Integer) **is**

**begin**

**if** C((K-1) mod 5)≠2 **and** C(K)=1 **and** C((K+1) mod 5)≠2 **then**

C(K):=2;

V(prisem(K));

*-- ^ gdy warunki zostają spełnione, udostępniamy miejsce*

**end if**;

**end** test;

Gdzie:

- C(0…4) to tablica zmiennych stanów filozofów, inicjalizowana na 0 - na początku wszyscy myślą

- prisem(0…4) to tablica semaforów dostępu do posiłku filozofów, inicjalizowana na 0

Następnie kod tego rozwiązania problemu prezentuje się następująco:

C: array(0…4) of Natural := (0,0,0,0,0)

prisem: array(0…4) of Natural := (0,0,0,0,0)

mutex: Natural := 1; *-- semafor zapewniający atomowość*

**task** Filozofi *-- jeden z 5 filozofów*

**task body** Filozofi **is**

**begin**

**loop**

Myśl;

P(mutex);

C(i):=1; *-- < sygnalizujemy chęć jedzenia*

test(i); *-- < sprawdzamy warunki*

V(mutex);

P(prisem(i));

Jedz;

P(mutex);

C(i):=0;

test((i+1) mod 5);

test((i-1) mod 5);

*-- ^ po posiłku, sprawdzamy czy sąsiedzi zgłodnieli*

V(mutex);

**end loop**;

**end** Filozofi;

Rozwiązanie to, jak poprzednie zapobiega sytuacji *zakleszczenia*, jednak nadal wrażliwe jest na *zagłodzenie*. Jednakże to rozwiązanie jest o tyle lepsze, że po stosunkowo niewielkiej analizie i wprowadzeniu odpowiednich zmian, można i temu problemowi zaradzić. Np. wprowadzając do zmiennych *C* dodatkowy stan „*bardzo głodny*”.