Gliwice, 18.06.2018

Obliczenia Równoległe

Temat:

Problem ucztujących filozofów

Bartłomiej Krasoń

Informatyka, sem. 4, gr. 6

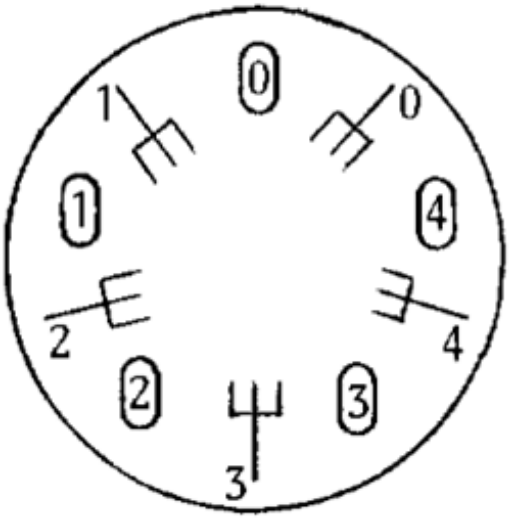
bartkra814@student.polsl.pl

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 Dijkstrę**, 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 algorytm *quicksort*. Problem ten wydaje się być niebanalny, gdyż jak widać, pochylają się 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.



Rys. 1 - Ilustracja, prezentująca graficznie „Problem ucztujących filozofów".

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 piątka 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 Filozof_i∈(0,1,2,3,4) -- jeden z 5 filozofów
task body Filozof_i is
begin
      loop
            Myśl;
            Jedz;
      end loop;
end Filozof_i;
```

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 Filozof_i∈(0,1,2,3,4) -- jeden z 5 filozofów
task body Filozof_i 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 Filozof_i;
```

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 (**ang. *deadlock***)**. 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 Filozof_{i∈(0,1,2,3,4)} -- jeden z 5 filozofów
task body Filozof_i 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 Filozof_i;
```

Takie podejście eliminuje problem wystąpienia *zakleszczenia*, jednakże nadal nie jest ono w stu procentach idealne. Może dojść do tak zwanego **zagłodzenia (***ang. livelock***)**. Jest to sytuacja, w której pewien 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 wywłaszczenie 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:

$$C[i] = 2 \textbf{ and } C[i + 1 \bmod 5] = 2$$

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 $C[i] = 1$
- filozof *i+1-wszy* nie je $C[i + 1] \neq 2$

- filozof *i-1-wszy* nie je $C[i-1] \neq 2$

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;
```

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`

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 Filozof_{i∈(0,1,2,3,4)} -- jeden z 5 filozofów
task body Filozof_i 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 Filozof_i;
```

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*".

Rozwiązanie asymetryczne

Jest to kolejne rozwiązanie korzystające z semaforów binarnych, jednakże od poprzednich różni się tym, że dzielimy filozofów na dwie grupy: *parzystych* oraz *nieparzystych*. Podział ten dotyczy procedur, jakie poszczególni filozofowie będą wykonywali, a różnicą w tych procedurach jest kolejność pobierania widelców. Tak więc:

- filozofowie *parzyści*: pierw podnoszą prawy widelec, następnie lewy
- filozofowie *nieparzyści*: pierw podnoszą lewy widelec, następnie prawy

```
Widelec: array(0…4) of Natural := (1,1,1,1,1)
task Parzysty_Filozof_i∈(0,2,4)
task body Parzysty_Filozof_i is
begin
      loop
            Myśl;
            P(Widelec((i+1) mod 5)); -- pobierz prawy widelec
            P(Widelec(i)); -- pobierz lewy widelec
            Jedz;
            V(Widelec(i)); -- zwolnij lewy widelec
            V(Widelec((i+1) mod 5)); -- zwolnij prawy widelec
end loop;
end Parzysty_Filozof_i;

task Nieparzysty_Filozof_i∈(1,3)
task body Nieparzysty_Filozof_i is
begin
      loop
            Myśl;
            P(Widelec(i)); -- pobierz lewy widelec
            P(Widelec((i+1) mod 5)); -- pobierz prawy widelec
            Jedz;
            V(Widelec((i+1) mod 5)); -- zwolnij prawy widelec
            V(Widelec(i)); -- zwolnij lewy widelec
      end loop;
end Nieparzysty_Filozof_i;
```

Takie rozwiązanie, eliminuje możliwość wystąpienia *zakleszczenia*, gdyż nie dopuszcza do sytuacji w której każdy filozof zostałby z jednym widelcem w ręku. Jednocześnie to rozwiązanie radzi sobie również z problemem *zagłodzenia*, gdyż filozof sygnalizując że jest głodny pobiera dostępny widelec i oczekuje na zwolnienie drugiego, co eliminuje problem zaistniały w powyższych rozwiązaniach, gdzie filozof musiał oczekiwać na moment, w którym oba widelce były jednocześnie dostępne.

Rozwiązanie z kelnerem

Ideą tego podejścia jest wprowadzenie zewnętrznego procesu nadrzędnego, który będzie pilnował przydzielania zasobów poszczególnym procesom – filozofom. W naszym przykładzie będzie to kelner krążący wokół stołu, oczekujący na sygnały od filozofów oznaczające chęć spożycia posiłku. Po odebraniu przez kelnera takiego zgłoszenia, kelner obsłuży danego filozofa, podaniem mu widelców leżących po jego lewej i prawej stronie, gdy tylko będą dostępne. Po wykonaniu zadania, kelner powróci do krążenia wokół stołu i nasłuchiwania na kolejne zgłoszenia. Najistotniejsze w tym podejściu jest to, że procesy podrzędne nie mogą bezpośrednio pobierać zasobów, tylko robią to przy pomocy procesu nadrzędnego – pilnującego. Zwalnianie zasób odbywa się bez udziału procesu nadrzędnego.

```
Widelec: array(0…4) of natural := (1,1,1,1,1)
Kelner: Natural := 1;
task Filozof_{i∈(0,1,2,3,4)}
task body Filozof_i is
begin
      loop
            Myśl;
            P(Kelner); -- zawołaj kelnera
            P(Widelec(i));
            P(Widelec((i+1) mod 5));
            V(Kelner);
            Jedz;
            V(Widelec(i));
            V(Widelec((i+1) mod 5));
            -- ^ odłóż widelce
      end loop;
end Filozof_i;
```

Powyższe podejście eliminuje całkowicie problemy *zakleszczenia* jak i *zagłodzenia*, co sprawia że jego działanie jest skuteczne. Jednakże nie należy ono do podejść najwydajniejszych, gdyż mogą wystąpić sytuacje, w których kelner będzie musiał spędzić znaczną część czasu przy jednym filozofie, w oczekiwaniu na zwolnienie dostępu do sąsiadujących z nim widelców – w trakcie gdy inni mogliby już podjąć się jedzenia.

Rozwiązanie z użyciem monitorów

Rozwiązanie to jest alternatywą dla użycia semaforów, które są narzędziem niskiego poziomu do rozwiązywania problemów synchronizacji. Monitor jest konstrukcją wyższego poziomu, który może być używany przez wiele procesów, jednakże jego poszczególne metody w danym momencie mogą być wykonywane tylko przez jeden proces, co zapewnia pożądane bezpieczeństwo. Monitor używany w naszym rozwiązaniu składa się z:

- tablicy *widelce(0…4)* – która przyjmuje wartość od 0 do 2 w zależności od tego, ile dostępnych widelców ma *i-ty* filozof
- tablica *zezwolenie(0…4)* – która umożliwia przesłanie sygnału *i-temu* filozofowi z pozwoleniem na przystąpienie do posiłku

- metoda *pobierz_widelce* – która zapewnia dostęp filozofowi do widelców, oczekuje aż dwa będą wolne, następnie aktualizuje informację o dostępności widelców w tablicy *widelce*
- metoda *zwróć_widelce* – która zapewnia zwrot widelców od filozofa, aktualizuje informacje w tablicy *widelce*

```
monitor Monitor_Widelcow is
      widelce: array(0..4) of Natural range 0..2 := {2,2,2,2,2};
      zezwolenie: array(0..4) of Condition;
      procedure pobierz_widelce(i: Natural) is
      begin
            if widelce(i) < 2 then
                  P(zezwolenie(i));
            end if;
            widelce((i+1) mod 5) := widelce((i+1) mod 5) - 1;
            widelce((i-1) mod 5) := widelce((i-1) mod 5) - 1;
      end pobierz_widelce;
      procedure zwroc_widelce(i: Natural) is
      begin
            widelce((i+1) mod 5) := widelce((i+1) mod 5) + 1;
            widelce((i-1) mod 5) := widelce((i-1) mod 5) + 1;
            if widelce((i+1)mod5) = 2 then
                  V(zezwolenie((i+1) mod 5));
            end if;
            if widelce((i-1)mod5) = 2 then
                  V(zezwolenie((i-1) mod 5));
            end if;
      end zwroc_widelce;
end Monitor_Widelcow;

task Filozof_i∈(0,1,2,3,4)
task body Filozof_i is
begin
      loop
            Myśl;
            pobierz_widelce(i);
            Jedz;
            zwroc_widelce(i);
      end loop;
end Filozof_i;
```

Rozwiązanie z użyciem monitorów jest uważane za bardziej eleganckie niż użycie semaforów, a to po części przez to, że jest one bliższe obiektowemu podejściu do programowania. Zauważamy że monitor zachowuje się jak swego rodzaju klasa, gdzie dostęp do jego prywatnych pól zapewniony jest tylko i wyłącznie poprzez udostępniane przez niego procedury. W podejściu tym nie występuje ryzyko *zakleszczenia* jednakże nie chroni ono w pełni przed zjawiskiem *zagłodzenia*.

Podsumowanie

Analiza rozwiązania *„Problemu ucztujących filozofów"* uzmysławia jak istotne jest zagwarantowanie odpowiedniej – przede wszystkim skutecznej – synchronizacji działania procesów współbieżnych w systemie. Ukazuje jak groźne mogą być niepożądane zjawiska takie jak *zakleszczenie* czy *zagłodzenie*. Warto również zwrócić uwagę na to, że stosunkowo łatwy problem, znajduje rozwiązanie w wielu różnych postaciach. Ciężko wskazać jedno, najlepsze rozwiązanie. Dopiero szczegółowa analiza problemu oraz założeń, jakie ma spełniać dany system może wskazać które rozwiązanie warto zastosować w tym przypadku. Gdy zależy nam na wydajności systemu i jesteśmy w stanie pozwolić sobie na zagłodzenie – jednym z lepszych rozwiązań okazuje się *rozwiązanie z użyciem monitorów*. Natomiast jeśli musimy wykluczyć w stu procentach zagłodzenie oraz zakleszczenie – dobry rozwiązaniem okazuje się *rozwiązanie z kelnerem*.

Kod programu w języku ADA

Rozwiązanie problemu ucztujących filozofów z wykorzystaniem kelnera:

```
--GNAT 4.9.3
with Ada.Text_IO; use Ada.Text_IO;

procedure Ucztujacy_Filozofowie is

      procedure Mysl(i: Natural) is
      begin
            Put_Line("Filozof "&Natural'Image(i)&" mysli");
      end Mysl;

      procedure Jedz(i: Natural) is
      begin
          Put_Line("Filozof "&Natural'Image(i)&" je");
      end Jedz;

      procedure P(i: in out Natural) is
      begin
          while i <= 0 loop
              delay 10.0;
          end loop;
          i:=i-1;
      end P;

      procedure V(i: in out Natural) is
      begin
          i:=i+1;
      end V;
```

```
      Widelec: array(0..4) of Natural := (others =>1);
      Kelner: Natural := 1;
      task type Filozof (index: Integer);
      task body Filozof is
      begin
          loop
              Mysl(index);
              P(Kelner);
                  P(Widelec(index));
                  P(Widelec((index+1) mod 5));
              V(Kelner);
              Jedz(index);
              V(Widelec(index));
              V(Widelec((index+1) mod 5));
          end loop;
      end Filozof;

      F0: Filozof (0);
      F1: Filozof (1);
      F2: Filozof (2);
      F3: Filozof (3);
      F4: Filozof (4);
begin
    Put_Line("start");
end Ucztujacy_Filozofowie;
```

Bibliografia

[1] Czech Zbigniew J: *Wprowadzenie do obliczeń równoległych*, Warszawa 2010

[2] Dijkstra E. W.: *Hierarchial Ordering of Sequential Processes*

[3] Ben-Ari M.: *Podstawy programowania współbieżnego i rozproszonego*, Warszawa 2009