1 Współbieżne wykonanie wielu procesów

Zadanie Uruchamiamy współbieżnie dwa egzemplarze następującego procesu:

```
var
  y : integer := 0;

process P1;
var
  x, i: integer;
begin
  for i := 1 to 5 do
  begin
    x := y;
    x := x + 1;
    y := x
  end;
end;
```

Jaką wartość będzia miała zmienna y po zakończeniu działania obu?

2 Algorytm Petersona

Zadanie Uruchamiamy współbieżnie dwa następujące procesy:

```
process P1;
begin
  repeat
    sekcja_lokalna;
    protokół_wstępny;
    rejon_krytyczny;
    protokół_końcowy
  until false
end;

process P2;
begin
  repeat
    sekcja_lokalna;
    protokół_wstępny;
    rejon_krytyczny;
    protokół_końcowy
  until false
end;
```

Chcemy zapewnić, że w tym samym czasie co najwyżej jeden z nich wykonuje fragment programu oznaczony jako `rejon_krytyczny`. Jakie instrukcje należy umieścić w protokołach, aby zrealizować ten cel? Zakładamy, że nie dysponujemy żadnymi mechanizmami synchronizacyjnymi.

Rozwiązanie W rozwiązaniu będziemy korzystać ze zmiennych globalnych i lokalnych. Zmienna lokalna może znajdować się w prywatnej przestrzeni adresowej procesu. Pozostałe procesy nie mają do niej dostępu, nie mogą jej zatem ani odczytywać ani modyfikować. Inaczej sytuacja wygląda ze zmiennymi globalnymi. Są one współdzielone między procesami, co oznacza, że w dowolnej chwili każdy z nich może takie zmienne zmodyfikować lub je odczytywać. Co dzieje się, gdy dwa wykonujące się równolegle procesy w tej samej chwili chcą uzyskać dostęp do tej samej zmiennej, a zatem do tej samej komórki (tych samych komórek) pamięci? Konflikt rozwiązuje sprzęt

za pomocą *arbitra pamięci*. Jest to układ sprzętowy, który realizuje wzajemne wykluczanie przy dostępie do pojedynczych komórek pamięci. Jednoczesne odwołania do tej samej komórki pamięci zostaną w jakiś nieznany z góry sposób uporządkowane w czasie i wykonane. W dalszej części rozważań zakładamy istnienie arbitra pamięci i jego poprawne działanie.

Spróbujmy najpierw rozwiązać problem wprowadzając zmienną globalną `ktoczeka`. Będzie ona przyjmować wartości 1 lub 2. Wartość 1 oznacza, że proces pierwszy musi teraz poczekać a prawo wejścia do rejonu krytycznego ma proces drugi. Treść procesów wygląda następująco:

```
var
  ktoczeka: 1..2 := ?;
process P1;
begin
  repeat
    sekcja_lokalna;
    while ktoczeka = 1 do
      { instrukcja pusta };
    rejon_krytyczny;
    ktoczeka := 1;
  until false
end;

process P2;
begin
  repeat
    sekcja_lokalna;
    while ktoczeka = 2 do
      { instrukcja pusta };
    rejon_krytyczny;
    ktoczeka := 2
  until false
end;
```

Czy jest to rozwiązanie poprawne? Własnośc *bezpieczeństwa* jest zachowana — nigdy oba procesy nie będą jednocześnie w rejonie krytycznym. A co z własnością żywotności? Przypomnijmy o założeniu, że proces nie przebywa nieskończenie długo w rejonie krytycznym. Zatem po wyjściu z niego (które na pewno w końcu nastąpi) rozpocznie się wykonanie `sekcji_lokalnej`. I tu pojawia się problem, bo o tym fragmencie programu nic założyć nie możemy. Jeśli proces utknie w tym fragmencie kodu (bo nastąpi błąd, zapętlenie, itp.) to drugi z procesów będzie mógł wejść do sekcji krytycznej jeszcze co najwyżej raz. Kolejna próba zakończy się wstrzymaniem procesu „na zawsze". Zatem przedstawione rozwiązanie nie ma własności żywotności. Inną wadą tego rozwiązania jest zbyt *ścisłe powiązanie* ze sobą procesów. Muszą one korzystać z sekcji krytycznej naprzemiennie, a przecież potrzeby obu procesów mogą być różne. Jeśli ponadto działają one z różną „szybkością" (bo na przykład sekcja lokalna jednego z nich jest bardziej złożona od sekcji lokalnej drugiego), to proces „szybszy" będzie równał tempo pracy do wolniejszego.

Spróbujmy zatem zaatakować problem inaczej. Wprowadźmy dwie logiczne zmienne globalne: `jest1` oznaczającą, że proces `P1` jest w sekcji krytycznej i analogiczną zmienną `jest2` dla procesu `P2`. Przed wejściem do rejonu krytycznego proces sprawdza, czy jego partner jest już w rejonie krytycznym. Jeśli tak, to czeka. Gdy rejon krytyczny będzie wolny to proces ustawia swoją zmienną sygnalizując, że jest w rejonie krytycznym, po czym wchodzi do niego.

```
var
  jest1: boolean := false;
  jest2: boolean := false;
```

```
process P1;
begin
  repeat
    sekcja_lokalna;
    while jest2 do
      { instrukcja pusta };
    jest1 := true;
    rejon_krytyczny;
    jest1 := false;
  until false
end;

process P2;
begin
  repeat
    sekcja_lokalna;
    while jest1 do
      { instrukcja pusta };
    jest2 := true;
    rejon_krytyczny;
    jest2 := false
  until false
end;
```

Zauważmy, że to rozwiązanie nie uzależnia już procesów od siebie. Jeśli jeden z nich nie chce korzystać z sekcji krytycznej lub awaryjnie zakończy swoje działanie w sekcji lokalnej, to drugi może swobodnie wchodzić do rejonu krytycznego, ile razy zechce. Nie ma też problemu z żywotnością. Jeśli jakiś proces utknął w pętli w protokole wstępnym, to drugi proces musi znajdować się gdzieś między przypisaniem `jest2 := true` a przypisaniem `jest2 := false`. Po skończonym czasie wyjdzie zatem z rejonu krytycznego i ustawi swoją zmienną `jest` na `false` pozwalając partnerowi wyjść z jałowej pętli i wejść do rejonu krytycznego. Niestety, przy pewnych złośliwych przeplotach może się zdarzyć, że do rejonu krytycznego wejdą oba procesy:

```
jest1 = false, jest2= false

P1:                                P2
while jest2 do
{warunek jest teraz falszywy
 więc nie czeka w pętli}
                                     while jest1 do
                                     {warunek jest teraz falszywy
                                      więc nie czeka w pętli}

jest1 := true;
rejon_krytyczny;
                                     jest2:= true;
                                     rejon_krytyczny;
{ oba procesy są w rejonie krytycznym }
```

Przyczyną takiej sytuacji było zbyt późne ustawienie zmiennych logicznych. Proces już był w rejonie krytycznym (bo przeszedł przez wstrzymującą go pętlę), a jeszcze nie poinformował partnera o tym, że jest w rejonie krytycznym. Zgodnie z definicją poprawności (musi być dobrze dla *każdego* przeplotu) stwierdzamy, że powyższy program jest niepoprawny. Spróbujmy zatem zmienić kolejność czynności i najpierw ustawmy zmienne logiczne, a potem próbujmy przechodzić przez pętle. Teraz zmienne logiczne oznaczają *chęć* wejścia do rejonu krytycznego:

```
var
  chce1: boolean := false;
  chce2: boolean := false;

process P1;
begin
  repeat
    sekcja_lokalna;
    chce1 := true;
    while chce2 do
      { instrukcja pusta };
    rejon_krytyczny;
    chce1 := false;
  until false
end;

process P2;
begin
  repeat
    sekcja_lokalna;
    chce2 := true;
    while chce1 do
      { instrukcja pusta };
    rejon_krytyczny;
    chce2 := false
  until false
end;
```

Teraz mamy program bezpieczny. Faktycznie w rejonie krytycznym może znajdować się co najwyżej jeden proces. Ale ... nie ma żywotności! Z łatwością można doprowadzić do zakleszczenia:

```
chce1 = false, chce2 = false

P1:                                P2
chce1 := true;
                                    chce2 := true;
while chce2 do;                     while chce1 do

{ i oba procesy są w pętlach nieskończonych }
```

Zatem znowu program jest niepoprawny. Można próbować ratować sytuację zmuszając procesy do chwilowej rezygnacji z wejścia do sekcji i ustąpienia pierwszeństwa partnerowi:

```
var
  chce1: boolean := false;
  chce2: boolean := false;

process P1;
begin
  repeat
    sekcja_lokalna;
    chce1 := true;
    while chce2 do
    begin
      chce1 := false;
```

```
      chce1 := true
    end;
    rejon_krytyczny;
    chce1 := false;
  until false
end;

process P2;
begin
  repeat
    sekcja_lokalna;
    chce2 := true;
    while chce1 do
    begin
      chce2 := false;
      chce2 := true
    end;
    rejon_krytyczny;
    chce2 := false
  until false
end;
```

Niestety znów istnieje (bardzo) złośliwy przeplot, który powoduje brak żywotności:

```
chce1 = false, chce2 = false

P1:                                P2
chce1 := true;
                                    chce2 := true;
while chce2 do
{warunek policzony i prawdziwy}
chce1:=false;
chce1 := true;
                                    while chce1 do
                                    {warunek policzony i prawdziwy}
                                    chce2 := false;
                                    chce2 := true
{ itd. }
```

Nie pomoże argumentacja, że taki przeplot jest „w zasadzie" nieprawdopodobny. Zgodnie z definicją poprawności, skoro *istnieje* scenariusz powodujący brak żywotności, to program jest niepoprawny.

Poprawne rozwiązanie znane pod nazwą *algorytmu Petersona* jest połączeniem pierwszego pomysłu z przedostatnim. Utrzymujemy zmienne `chce1` i `chce2`, które oznaczają chęć wejścia procesu do rejonu krytycznego. W razie gdy oba procesy chcą wejść do rejonu krytycznego rozstrzygamy konflikt za pomocą zmiennej `ktoczeka`:

```
var
  chce1: boolean := false;
  chce2: boolean := false;
  ktoczeka: 1..2 := ?

process P1;
begin
  repeat
```

```
    sekcja_lokalna;
    chce1 := true;
    ktoczeka := 1;
    while chce2 and (ktoczeka = 1) do
      { instrukcja pusta };
    rejon_krytyczny;
    chce1 := false;
  until false
end;

process P2;
begin
  repeat
    sekcja_lokalna;
    chce2 := true;
    ktoczeka := 2;
    while chce1 and (ktoczeka = 2) do
      { instrukcja pusta };
    rejon_krytyczny;
    chce2 := false
  until false
end;
```

Wady algorytmu Petersona:

1. Aktywne oczekiwanie. Z założenia nie było dostępnych żadnych mechanizmów wstrzymywania procesów, nie pozostało nic innego niż zatrzymanie procesu w jałowej pętli. Angażuje to jednak czas procesora (i szynę danych). W przyszłości aktywne oczekiwanie będziemy traktować jak poważny błąd.

2. Liczba procesów musi być znana z góry.

3. Koszt protokołu wstępnego jest znaczny.