Programowanie współbieżne
Zadanie nr 4
Spotkania

Cel zadania.

Celem zadania jest zapoznanie się mechanizmem spotkań służącym do synchronizacji i komunikacji zadań.

Treść zadania.

Należy zorganizować współpracę procesów na podstawie następujących założeń:
- Procesy należy podzielić na dwie klasy – procesy symulujące zadania wywołujące i procesy symulujące zadania przyjmujące.
- Zadania wywołujące powinny wykonywać usługi udostępniane im przez zadania przyjmujące posługując się mechanizmem spotkania.
- Implementacja mechanizmu spotkania powinna być maksymalnie zgodna, co do semantyki, z definicją tegoż mechanizmu przedstawioną poniżej.
- Implementacja spotkania powinna zawierać implementację instrukcji **select** po stronie zadania przyjmującego

Podstawa zaliczenia.

1) Zademonstrowanie działającego programu.
2) Przedstawienie sprawozdania zawierającego:
   - krótki opis zastosowanego rozwiązania problemu.
   - udowodnienie, że zastosowane rozwiązanie jest poprawne.

Pytania kontrolne.

1) Jaka jest podstawowa różnica pomiędzy mechanizmem monitora a mechanizmem spotkania?
2) Przedstaw algorytm mechanizmu spotkania.
3) Na czym polega asymetria identyfikacji zadań dla mechanizmu spotkania?
4) Omów semantykę instrukcji select
5) Wskaż i omów różnie pomiędzy zaproponowaną przez Ciebie implementacją spotkania i instrukcji select , a zdefiniowanymi w języku Ada semantykami tych mechanizmów.

Część teoretyczna.

1) Zagadnienia implementacyjne

Spotkanie jest mechanizmem komunikacji pomiędzy zadaniami (procesami) zaproponowanym przez twórców języka Ada jako pierwotny mechanizm synchronizacji zadań. Nazwa „spotkanie" jest dobrą analogią rzeczywistych cech tego mechanizmu, który w naturalny sposób pozwala na prostą implementację systemów typu klient – serwer. Aby doszło do komunikacji dwóch zadań musi nastąpić ich spotkanie, przy czym jedno z zadań, zadanie wywołujące (klient), pragnie odwołać się do pewnej usługi oferowanej przez zadanie przyjmujące (serwer). Obydwa zadania wykonują się współbieżnie.
Zadanie wywołujące zna nazwę zadania przyjmującego oraz nazwę usługi do której pragnie się odwołać, zadanie przyjmujące „wie" jedynie, że któreś z zadań wywołujących pragnie się odwołać do zadanej usługi przezeń oferowanej.

Implementacja spotkania od strony zadania wywołującego przypomina do złudzenia zwykłe wywołanie procedury lub funkcji. Zadanie to wywołuje funkcję serwera i zostaje wstrzymane do momentu realizacji przez serwer żądanej usługi. Od strony zadania przyjmującego, implementacja jest bardziej skomplikowana ze względu na fakt, że jest ono w tym przypadku osobnym procesem. Z każdą usługą oferowaną przez zadanie przyjmujące skojarzona jest osobna kolejka FIFO zadań wywołujących, które oczekują na wykonanie wspomnianej usługi. Rolą zadania przyjmującego jest przepatrywanie kolejek zadań wywołujących skojarzonych z zadanymi usługami i w miarę możliwości wykonywanie tych usług na rzecz poszczególnych zadań wywołujących, które znajdują się aktualnie w kolejkach zadań oczekujących.

Przyjrzyjmy się definicji spotkania w języku Ada zwracając szczególną uwagę na semantykę spotkania oraz semantykę instrukcji **select**.

2) Teoria spotkań – język Ada

Komunikacja w Adzie jest synchroniczna (i niebuforowana). Aby doszło do komunikacji dwu zadań[1] ([1] W tym rozdziale procesy będziemy nazywać zadaniami, aby zachować zgodność z terminologią stosowaną w Adzie.) musi nastąpić ich spotkanie (ang. *rendezyous).* Ta nazwa ma przywoływać obraz dwu osób, które ustaliły miejsce swojego spotkania (rys. 8. 1 (a)). Ta, która przyjdzie pierwsza, musi czekać na przyjście drugiej.

W tym obrazie rola obu osób jest identyczna, a miejsce spotkanie neutralne. W Adzie natomiast miejsce spotkania należy do jednego z zadań, zwanego zadaniem *przyjmującym.* Drugie zadanie, zadanie wywołujące, musi znać nazwę zadania przyjmującego oraz nazwę miejsca spotkania (rys. 8. 1 (b)), zwanego wejściem. Zadanie przyjmujące natomiast nie zna nazwy zadania wywołującego. Istotne jest podkreślenie, że oba zadania wykonują się współbieżnie, a jedynym sposobem ich zsynchronizowania są spotkania.

Zaletą modelu asymetrycznego jest łatwość programowania serwerów. Zadanie przyjmujące może być np. serwerem drukarki, gotowym do przyjęcia żądań drukowania od dowolnego innego zadania w systemie. Gdyby przyjąć symetryczny system nazywania, to treść zadania będącego serwerem musiałaby być zmieniana za każdym razem przy dodawaniu nowego zadania.

Zajmijmy się teraz definicją spotkania w Adzie. Zadanie składa się z dwu części: *specyfikacji i treści.* Specyfikacja może zawierać jedynie deklaracje *wejść*[2] ([2]Dotychczas nie używaliśmy wejść, więc pomijaliśmy specyfikacje zadań. Każde zadanie musi jednak mieć specyfikację, nawet jeśli jest ona pusta.). Składniowo wejście wygląda dokładnie tak jak deklaracja procedury. Zrobiono tak po to, by umożliwić zamianę procedury sekwencyjnej na współbieżne wejście bez konieczności jakiegokolwiek innego zmieniania programu.

```
task Bufor is
entry Wstaw (I: in Integer) ;
entry Pobierz (I: out Integer) ;
end Bufor;
```

Gdy już zadeklarujemy specyfikację zadania i uczynimy ją widoczną dla pozostałych zadań, będą one mogły wywoływać wejścia używając notacji kropkowej - nazwy zadania, po której następuje nazwa wejścia i odpowiednie parametry, np.

```
Bufor.Wstaw(I) ;
```

Gdyby Wstaw było procedurą, sterowanie zostałoby natychmiast przekazane do treści procedury. Wywołanie wejścia musi się spotkać z instrukcją accept z zadania, w którym znajduje się to wejście:

```
task body Bufor is
begin
. . .
        accept Wstaw (I: in Integer) do
        . . .instrukcje stanowiące treść accept
        end Wstaw;
end Bufor;
```

Pamiętajmy, że zadanie przyjmujące jest zwykłym procesem sekwencyjnym wykonującym swe instrukcje. Tak jak i inne instrukcje, instrukcja accept jest wykonywana wtedy, gdy dojdzie do niej licznik instrukcji, z tym tylko wyjątkiem, że jej definicja wymaga synchronizacji z zadaniem wywołującym. W zakładanym przeplecie ciągów wykonań instrukcji albo zadanie wywołujące dojdzie do wywołania wejścia, zanim zadanie przyjmujące osiągnie instrukcję accept dla tego wejścia, albo odwrotnie. Którekolwiek z zadań będzie pierwsze, zostanie wstrzymane, aż drugie dojdzie do odpowiadającej instrukcji. W tym momencie zaczyna się spotkanie. Semantyka spotkania jest następująca.

(1) Zadanie wywołujące przekazuje swoje parametry wejściowe (in) do zadania przyjmującego i zostaje zawieszone do momentu zakończenia spotkania.

(2) Zadanie przyjmujące wykonuje instrukcje z *treści* accept (instrukcje znajdujące się po słowie kluczowym aż do odpowiadającego mu end).

(3) Parametry wyjściowe (out) są przekazywane do zadania wywołującego.

(4) Spotkanie jest teraz zakończone i żaden z procesów nie jest już wstrzymany.

Zauważmy, że zakończenie spotkania nie musi koniecznie powodować, że zadanie od razu zacznie się wykonywać. Decyzję o tym, które zadanie będzie wykonywane, pozostawia się zarządcy procesów. Zakończenie spotkania kończy wykonanie instrukcji accept. Jeśli zadanie chce wziąć udział w następnym spotkaniu, to musi wykonać następną instrukcję accept. Typowe niekończące się zadanie, takie jak serwer, będzie zatem miało instrukcję accept wewnątrz pętli, tak by mogło wziąć udział w wielu spotkaniach.

Rysunek 8.2 demonstruje instrukcję accept w programie, który zarządza zdegenerowanym, ograniczonym buforem. Zadanie producenta będzie wywoływać wejście Wstaw, a zadanie konsumenta wejście Pobierz. Instrukcje accept są umieszczone w pętli, tak więc po każdej parze spotkań (producenta z buforem, a potem konsumenta z buforem) zadanie bufora ponownie wykona instrukcję accept Wstaw.

Porównując to rozwiązanie z rozwiązaniem używającym monitora zauważamy, że bufor stał się oddzielnym zadaniem, a nie tylko pasywnym zbiorem danych i procedur. Jednak mimo narzutu spowodowanego dodatkowym zadaniem, rozwiązanie z Ady w implementacji wieloprocesorowej mogłoby być efektywniejsze, gdyż tylko część przetwarzania musi być wykonywana przy wzajemnym wykluczaniu zapewnianym przez spotkanie. W tym rozwiązaniu modyfikowanie indeksów i licznika nie musi być wykonywane w ramach spotkania, gdyż są to zmienne lokalne. Taka struktura jest typowa dla algorytmów zapisywanych w Adzie, gdyż synchronizacja jest zdefiniowana bezpośrednio między zadaniami, a nie za pośrednictwem systemu operacyjnego.

```
task body Bufor is
        B: array(0 ..N- l) of Integer;
        We_ind, Wy.ind: Integer := O;
        Licznik: Integer := 0;

        begin
            loop
                accept Wstaw( I: in Integer) do
                  B(We_ind) : = I;
                end Wstaw;
                Licznik : = Licznik + l;
                We_ptr : =  (We_ptr + l) mod N;
                accept Pobierz(I: out Integer) do
                I := B(Wy_Ind) ;
                end Pobierz;
                Licznik : = Licznik - l;
                Wy_ind : = (Wy_ind + l) mod N;
           end loop;
        end Bufor;
```

Rys. *8.2.* Zdegenerowany, ograniczony bufor

Wejścia należą do zadania, które je zadeklarowało, i tylko ono może wykonywać instrukcje accept przyjmujące ich wywołania. Jeżeli zadanie przyjmujące może mieć wiele instrukcji accept (nawet dla tego samego wejścia), to oczywiście może wykonywać na raz tylko jedną instrukcję. Kilka zadań może jednak wywoływać to samo wejście z innego zadania, na przykład kilku producentów mogłoby jednocześnie wywołać Bufor. Wstaw. Zadania wywołujące to samo wejście muszą ustawić się w kolejce, aby dojść do wejścia. Każda instrukcja accept powoduje spotkanie z pierwszym zadaniem czekającym w kolejce do tego wejścia. Zakończenie treści accept kończy spotkanie; zadanie przyjmujące nie kontynuuje spotkań z kolejnymi zadaniami czekającymi w kolejce do tego wejścia. Musi natomiast wykonać kolejną instrukcję accept dla tego wejścia (tzn. następne wystąpienie instrukcji accept, które może być tą samą instrukcją napotkaną w kolejnym obrocie pętli).

Podsumowując, spotkania w Adzie są mechanizmem pierwotnym o następującej charakterystyce:

- synchroniczna, niebuforowana komunikacja;
- asymetryczne identyfikowanie zadań - nadawca zna odbiorcę, ale odbiorca nadawcy już nie;
- dwukierunkowy przepływ danych podczas pojedynczego spotkania.

Zmienne globalne dzielone między wieloma zadaniami są także dozwolone w Adzie, zainteresowanych opisem tej możliwości odsyłamy do odpowiednich podręczników. Używaliśmy takich zmiennych do zaprogramowania w Adzie przykładów z rozdz. 3.

Instrukcja *select*

Rozwiązanie z rys. 8.2 jest zdegenerowane, ponieważ nie pozwala odmówić spotkania, gdy bufor jest pełny lub pusty. Ponadto wymusza ścisłe przeplatanie produkowania i konsumowania, co nie odpowiada definicji bufora. Przedstawiamy teraz instrukcję select pozwalającą zadaniu wybrać wywołanie wejścia do wykonania spośród kilku możliwości, a także warunkowo przyjmować wywołanie wejścia. Zanim opiszemy instrukcję selecfc, spójrzmy na poprawne rozwiązanie problemu ograniczonego bufora (rys. 8.3).

Instrukcja select pozwala buforowi niedeterministycznie wybrać jedną z dwu *dozorowanych* gałęzi. Dozory są wyrażeniami logicznymi poprzedzającymi instrukcje accept. Jeśli to wyrażenie ma wartość prawda, to odpowiadającą mu gałąź (ang. *alternative)* instrukcji select nazywamy *otwartą,* a spotkanie w instrukcji accept jest dozwolone. Jeśli wartością wyrażenia jest fałsz, to mówimy, że odpowiadająca gałąź jest *zamknięta,* spotkanie zaś nie jest dozwolone.

Jeśli w przedstawionym przykładzie bufor jest pusty, czyli *Licznik* = O, to instrukcja selecfc zredukuje się do zwykłej instrukcji accept z otwartej gałęzi Wstaw. Jeśli bufor jest pełny, czyli *Licznik* = *N, to* również będzie tylko jedna otwarta gałąź instrukcji select. Warto zauważyć, że zawsze co najmniej jedna gałąź będzie otwarta, gdyż jest niemożliwe, aby jednocześnie zachodziło[3] *(*3 Zakładając, jak zwykle, że rozmiar bufora jest większy od zera*.) Licznik* = O i *Licznik* = *N.*

```
task body Bufor is
        B: array(0..N-1) of Integer;
        We_ind, Wy_ind: Integer := O;
        Licznik: Integer := 0;
begin
  loop
        select
          when Licznik < N =>
          accept Wstaw(I: in Integer) do
          B(We_ind) : = I;
           end Wstaw;
          Licznik : = Licznik + l;
        We_ind : = (We_ind + l) mod N;
        or
        when Licznik > O =>
                accept Pobierz(I: out Integer) do
                I : = B(Wy_Ind) ;
        end Pobierz;
        Licznik : = Licznik – l;
        Wy_ind : = (Wy_ind + l) mod N;
        end select;
  end loop;
end Bufor;
```

Rys. *8.3.* Bufor ograniczony

Jeśli O < *Licznik* < *N,* to obie gałęzie są otwarte. Jeśli nie ma zadań oczekujących w kolejkach wejściowych, to zadanie przyjmujące będzie czekać na pierwsze zadanie, które wywoła dowolne z wejść. W przeciwieństwie do algorytmu z poprzedniego podrozdziału, który wymagał wywołania od konsumenta po każdym wywołaniu od producenta, użycie instrukcji select pozwoliło buforowi spotkać się z tym spośród zadań, które wywoła go jako pierwsze.

Jeśli zadania wywołujące czekają tylko w jednej z kolejek wejściowych, to spotkanie odbędzie się z pierwszym zadaniem z tej kolejki. Jeśli jednak w obu kolejkach wejściowych są oczekujące zadania, to zadanie przyjmujące wybierze niedeterministycznie do spotkania się pierwsze zadanie z jednej z dwu kolejek. Rozumiemy przez to, że implementacja ma prawo wybrać jedną z otwartych gałęzi zgodnie z dowolnym (rozsądnym) algorytmem. Wybór niedeterministyczny nie oznacza losowości, która implikuje, że nie da się przewidzieć, jaki wybór zostanie dokonany. Implementacja wybierająca zawsze pierwszą otwartą gałąź jest poprawną implementacją. Z punktu widzenia programisty niedeterminizm oznacza, że poprawność programu nie może zależeć od tego, jaki algorytm będzie użyty do wyboru jednej z wielu otwartych gałęzi. Jest tak w przedstawionym rozwiązaniu problemu ograniczonego bufora. Gdyby jedna z gałęzi stale była faworyzowana, to w

końcu bufor by się wypełnił (lub stał się pusty), faworyzowana gałąź byłaby zamknięta i musiałaby być wybrana druga.

Podamy teraz pełną definicję instrukcji select. Składniowo instrukcja select zawiera dowolną liczbę dozorowanych instrukcji accept oddzielonych zarezerwowanym słowem or. Dozór można pominąć, co jest równoważne napisaniu when True ==>. Po każdej instrukcji accept można napisać ciąg instrukcji. Ostatnia gałąź instrukcji select może przyjąć jedną z następujących postaci:

- else, po którym następuje ciąg instrukcji;
- delay T, po którym następuje ciąg instrukcji, przy czym T jest wyrażeniem o wartościach typu real;
- terminale.

Te specjalne postacie gałęzi wzajemnie się wykluczają, to znaczy, że może wystąpić tylko jedna z nich.

Semantyka instrukcji select jest następująca:

(1) Wylicza się dozory. Zbiór gałęzi, których dozory mają wartość prawda, nazywamy zbiorem gałęzi otwartych. Sytuacja, w której zbiór gałęzi otwartych jest pusty, jest błędem (chyba że jest gałąź else).

(2) Jeśli w kolejkach wejściowych do gałęzi otwartych czekają zadania wywołujące, to rozpocznie się spotkanie z pierwszym zadaniem z jednej z tych kolejek.

(3) Jeśli wszystkie kolejki do gałęzi otwartych są puste, to zadanie przyjmujące zostaje wstrzymane. Gdy tylko jakieś zadanie wywoła wejście ze zbioru gałęzi otwartych, to rozpocznie się spotkanie z tym zadaniem wywołującym. Zauważmy, że zbiór gałęzi otwartych nie zmienia się, ponieważ dozory nie są ponownie wyliczane w trakcie wykonywania instrukcji select.

(4) Spotkanie przebiega następująco:

- parametry wejściowe (in) są przekazywane od zadania wywołującego do zadania przyjmującego;
- zadanie wywołujące zostaje wstrzymane;
- zadanie przyjmujące wykonuje instrukcje stanowiące treść instrukcji accept;
- parametry wyjściowe (out) są przekazywane od zadania przyjmującego do zadania wywołującego;
- zadanie wywołujące jest wznawiane.

(5) Po zakończeniu spotkania zadanie przyjmujące wykonuje instrukcje znajdujące się za instrukcją accept.

(6) Instrukcja select z gałęzią else. Jeśli nie ma gałęzi otwartych lub są, ale nie ma zadań wywołujących w żadnej z ich kolejek wejściowych, to jest wykonywany ciąg instrukcji z gałęzi else.

(7) Instrukcja select z gałęzią delay. Jeśli nie ma zadań wywołujących w kolejkach wejściowych do gałęzi otwartych, to zadanie akceptujące będzie czekać, jak opisano, ale tylko przez czas zadany w klauzuli delay. Po jego upływie wykona się sekwencja instrukcji z gałęzi delay. Należy podkreślić, że zadanie przyjmujące nie musi być wznowione *dokładnie* po upływie tego okresu, lecz jeśli po tym czasie kolejki wejściowe nadal będą puste, to zadanie jest gotowe do wykonywania i będzie wykonywać instrukcje z gałęzi delay.

(8) Instrukcja select z gałęzią terminale. Pełne wyjaśnienie wykracza poza zakres książki. Z grubsza rzecz biorąc, jeśli ta instrukcja select jest zawieszona i wszystkie zadania, które mogłyby ewentualnie wywołać jej wejścia, zakończyły się lub również oczekują w instrukcji select z gałęzią terminale, to wówczas wszystkie zadania z tego zbioru kończą się.

Jak można zauważyć, instrukcja select jest mocnym i jednocześnie elementarnym mechanizmem pierwotnym programowania współbieżnego. Wykonuje dość efektywną operację niskopoziomową: sprawdza czy są wywołania i wybiera jedno z nich, ale różnorodne warianty tej instrukcji umożliwiają wyrażanie wielu algorytmów w naturalny sposób.

Literatura.

M.Ben-Ari „Podstawy programowania współbieżnego i rozproszonego” WNT 1996

Spis wariantów zadań.

Treść zadań jest identyczna z treścią zadań ćwiczenia nr 3 dotyczącego monitorów.