

Ada95: Procesy współbieżne

Ada95 oferuje dwie podstawowe formy komunikacji pomiędzy procesami:

- komunikację synchroniczną między parą zadań – spotkanie (*rendez-vous*),
- komunikację asynchroniczną między wieloma zadaniami – za pośrednictwem obiektów chronionych.

Zadania

Zadania są jednostkami strukturalizacji programów współbieżnych. Zadania mogą wykonywać się równolegle, przy czym sposób realizacji zrównoleglenia zależy od środowiska wykonawczego.

Zadanie jest obiektem *typu zadaniowego*. W programie można utworzyć wiele zadań na podstawie tego samego typu zadaniowego. Wszystkie obiekty danego typu mają jednakową *specyfikację* (interfejs) i tę samą *treść* (ciało, implementację). Obiekty zadaniowe są tworzone tak jak inne obiekty języka: mogą być deklarowane statycznie lub kreowane dynamicznie.

Deklarowanie zadań

Deklaracja zadania:

```
task Zad is
...
  entry E1(par1: in typ1; par2: out typ2; ...);
...
private
  entry E2(...);
...
end Zad;

task body Zad is
...
accept E1(par1: in typ1; par2: out typ2; ...)
do
...
  end E1;
...
accept E2(...)
do
...
  end E2;
...
end Zad;
```

wejście E2 deklarowane w części prywatnej jest niewidoczne dla innych zadań

Deklaracja typu zadaniowego:

```
task type T is
...
  entry E1(...);
...
end T;
```

Zadania mogą być elementami innych złożonych typów danych:

```
tablica: array(1..5) of T;

type R is
record
  Zadanie: T;
...
end record;
...
R1, R2 : R;
```

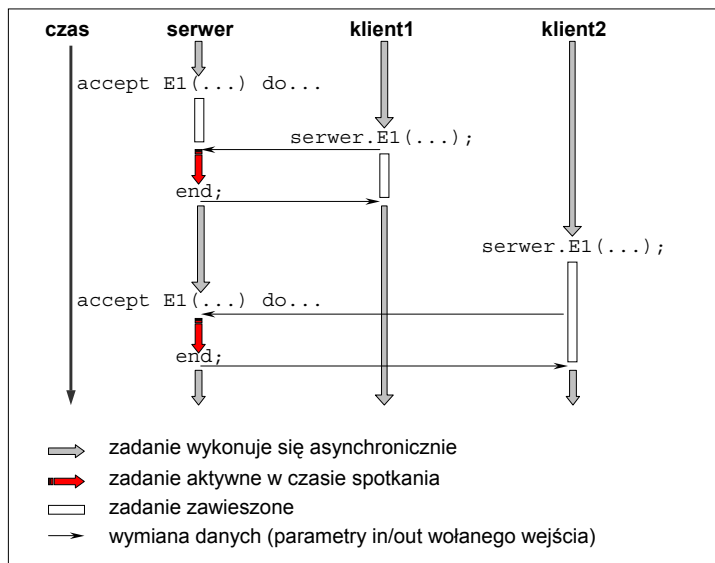
Spotkania asymetryczne

W komunikacji między zadaniami z wykorzystaniem mechanizmu spotkań istotne jest wyróżnienie roli jaką może w danej chwili pełnić zadanie. W różnych momentach czasu zadanie może być bierne (*serwer*) – jeśli udostępnia lub jest gotowe udostępnić usługi identyfikowane przez nazwy wejść – lub czynne (*klient*) – jeśli wywołuje właśnie wejście jakiegoś serwera.

Spotkanie między dwoma zadaniami jest następstwem wywołania przez jedno z zadań (klienta) wejścia drugiego zadania (serwera):

```
Nazwa_zadania.wejście(parametr1, ...);
```

Komunikacja pomiędzy oboma zadaniami jest możliwa dzięki wspólnej realizacji dwóch skojarzonych instrukcji: wywołania wejścia po stronie klienta i obsługi wejścia (*accept*) po stronie serwera. W zależności od tego, która instrukcja wykonana została wcześniej, jedno z zadań zostaje zawieszone do momentu rozpoczęcia spotkania.



Przykład: producent konsument

Jednoelementowy bufor pośredniczący między zadaniem producenta a zadaniem konsumenta:

```
task buffer is
  entry store(x: Produkt);
  entry remove(y: Produkt);
end buffer;

task body buffer is
  temp: Produkt;
begin
  loop
    accept store(x: Produkt);
    temp := x;
  end store;
  accept remove(y: Produkt);
  y := temp;
  end remove;
  end loop;
end buffer;
```

Zadania klienci:

```
task type Producent;

task body Producent is
  element: Produkt;
begin
  loop
    ...
    buffer.store(element);
    ...
  end loop;
end Producent;
```

```
task type Konsument;

task body Konsument is
  element: Produkt;
begin
  loop
    ...
    buffer.remove(element);
    ...
  end loop;
end Konsument;
```

```
Prod: Producent;
Kons: Konsument;
```

Kolejki wejść

Każdemu wejściu zadania wywoływanego (buffer) jest przyporządkowana kolejka zadań wywołujących. Domyślną strategią obsługi wejść jest kolejka FIFO.

Pragma Queuing_Policy

Pragma kompilacji `Queuing_Policy(...)` pozwala ustalić inną strategię, która będzie obowiązywać wszystkie zadania danego programu (o ile taka strategia jest dostępna w konkretnej implementacji języka).

Obiekty chronione

Obiekt chroniony jest jednostką programową, która organizuje dostęp zadań do grupowanych przez siebie danych współdzielonych. Budowa obiektu chronionego jest podobna do budowy pakietu i zadania – składa się ze specyfikacji i treści implementującej obiekt. Możliwe jest również definiowanie typów chronionych.

Specyfikacja obiektu (oraz typu) chronionego zawsze zawiera część publiczną i część prywatną. W ogólnym przypadku, publiczną część tworzą deklaracje funkcji, procedur oraz wejść. W części prywatnej występują deklaracje zmiennych współdzielonych i, opcjonalnie, deklaracje wewnętrznych funkcji, procedur i wejść.

Dostęp do obiektu chronionego jest możliwy tylko poprzez wywołania funkcji, procedur i wejść publicznych i odbywa się on zgodnie z zasadą **wzajemnego wykluczania**.

Wywołania funkcji pozwalają tylko na odczyt danych współdzielonych (podanych w części `private`), a wywołania procedur i wejść – na ich modyfikowanie.

```
protected zmienna_chroniona is
  function czytaj return zapis;
  procedure pisz (x: in zapis);
private
  element: zapis; -- zmienna współdzielona
end zmienna_chroniona;

protected body zmienna_chroniona is
  function czytaj return zapis is
  begin
    return element;
  end czytaj;
  procedure pisz (x: in zapis) is
  begin
    element:=x;
  end pisz;
end zmienna_chroniona;
```

W części implementacyjnej, z każdym z zadeklarowanych wejść jest związany warunek wykonania wejścia, nazywany barierą (*barrier*). Treść wołanego wejścia jest wykonywana tylko jeśli warunek bariery jest spełniony.

```
subtype Rozmiar is Integer range 1..Rozmiar_MAX;

protected type Bufor_cykliczny(N: Rozmiar:=100) is
  entry Put(x: in Wartość);
  entry Get(x: out Wartość);
private
  bufor: array(0..N-1) of Wartość; -- bufor N-elementowy
  put_ptr: Integer :=0;           -- indeks miejsca wstawiania
  get_ptr: Integer :=0;           -- indeks miejsca pobierania
  licznik: Integer range 0..N :=0 -- wypełnienie bufora
end Bufor_cykliczny;

protected body Bufor_cykliczny is
  entry Put(x: in Wartość) when licznik<N is
  begin
    bufor(put_ptr):=X;
    put_ptr:=(put_ptr+1) mod N;
    licznik:=licznik+1;
  end Put;
  entry Get(x: out Wartość) when licznik>0 is
  begin
    X:=bufor(get_ptr);
    get_ptr:=(get_ptr+1) mod N;
    licznik:=licznik-1;
  end Get;
end Bufor_cykliczny;
```

Blokady

W momencie, gdy zadanie wywołuje wejście lub procedurę obiektu chronionego, ten może być zajęty obsługą innego wywołania (*zablokowany*). Z każdym obiektem chronionym związane są dwie blokady: do czytania (*shared read lock*) – aktywna gdy obiekt chroniony obsługuje wywołanie swojej funkcji – i do pisania (*exclusive read/write lock*) – gdy obiekt obsługuje wywołanie procedury lub wejścia.

Reguły dostępu do obiektu chronionego:

- Jeżeli obiekt chroniony ma założoną blokadę *read* i wywoływana jest jego funkcja, to funkcja ta zostaje wykonana.
- Jeżeli obiekt chroniony ma założoną blokadę *read* i wywoływane jest jego wejście lub procedura, to wywołanie jest opóźniane, dopóki są zadania aktywne wewnątrz obiektu chronionego.
- Jeżeli obiekt chroniony ma założoną blokadę *read/write*, to wywołanie jest opóźniane, dopóki są zadania aktywne wewnątrz obiektu chronionego.
- Jeżeli nadchodzi kolej wykonania wywoływanego wejścia obiektu chronionego lecz bariera ma wartość `False`, to wywołanie jest ustawiane w kolejce związanej z tą barierą czekając na spełnienie warunku bariery; obiekt nie zostaje jeszcze zablokowany.

Złożone schematy komunikacji

Instrukcja select

Instrukcja `select` występuje w czterech podstawowych formach:

1. oczekiwanie selektywne

umożliwia zadaniu serwera:

- oczekiwanie na więcej niż jedno spotkanie,
- oczekiwanie na rozpoczęcie spotkania w ustalonym odcinku czasu,
- wycofanie oferty spotkania, jeżeli nie może ono nastąpić natychmiast,
- zakończenie istnienia zadania, jeżeli nie istnieją klienci, którzy wywołują jego wejścia

2. terminowe wywołanie wejścia

umożliwia zadaniu klienta:

- oczekiwanie na rozpoczęcie spotkania w ustalonym odcinku czasu,

3. warunkowe wywołanie wejścia

umożliwia zadaniu klienta:

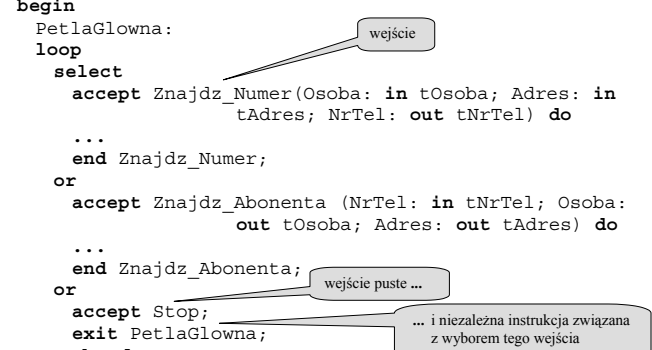
- wycofanie oferty spotkania, jeżeli nie może ono nastąpić natychmiast

4. asynchroniczna zmiana wątku sterowania

Oczekiwanie selektywne


```
task BiuroNumerow is
  entry Znajdz_Numer(...);
  entry Znajdz_Abonenta(...);
  entry Stop;
end BiuroNumerow;

task body BiuroNumerow is
  ...
begin
  PetlaGlowna:
  loop
    select
      accept Znajdz_Numer(Osoba: in tOsoba; Adres: in
                          tAdres; NrTel: out tNrTel) do
        ...
      end Znajdz_Numer;
    or
      accept Znajdz_Abonenta (NrTel: in tNrTel; Osoba:
                              out tOsoba; Adres: out tAdres) do
        ...
      end Znajdz_Abonenta;
    or
      accept Stop;
      exit PetlaGlowna;
    end select;
  end loop PetlaGlowna;
end BiuroNumerow;
```



Gałąź accept z dozorami

```
task body BiuroNumerow is
  ...
begin
  PetlaGlowna:
  loop
    select
      accept Znajdz_Numer(Osoba: in tOsoba; Adres: in
                          tAdres; NrTel: out tNrTel) do
        ...
      end Znajdz_Numer;
    or
      accept Znajdz_Abonenta (NrTel: in tNrTel; Osoba:
                              out tOsoba; Adres: out tAdres) do
        ...
      end Znajdz_Abonenta;
    or
      when MonterzyWolni > 0 =>
      accept Zgloszenie_Awarii (NrTel: in tNrTel) do
        ...
      end Zgloszenie_Awarii;
    or
      accept Stop;
      exit PetlaGlowna;
    end select;
  end loop PetlaGlowna;
end BiuroNumerow;
```



Przy braku *pragm* przyjmowana jest strategia FIFO obsługi kolejki żądań. Jeśli w momencie osiągnięcia instrukcji `select` zadania-klienci czekają w kolejkach kilku wejść, to wybór kolejki **jest niedeterministyczny**.

Każda gałąź `accept` instrukcji `select` może być poprzedzona wyrażeniem logicznym, nazywanym dozorem (*guard*). Wykonanie instrukcji `select` rozpoczyna się od obliczenia dozorów wszystkich gałęzi. Gałęzie, dla których dozór jest spełniony, są nazywane gałęziami otwartymi (pozostałe – zamkniętymi). Tylko gałęzie otwarte brane są pod uwagę w trakcie dalszego wykonania instrukcji `select`.

Gałęzie bez dozorów równoważne są gałęziom z dozorami `True`.

Wartości wszystkich dozorów obliczane są każdorazowo tylko raz na początku wykonania instrukcji `select` – tzn. jeżeli w wyrażeniu występują zmienne globalne, których wartości mogą ulec zmianie w innej gałęzi, to zmiany te nie zostaną zauważone, aż do ponownego wykonania instrukcji `select`.

Jeśli wszystkie gałęzie chronione są dozorami, a ich wartości są równe `False`, to generowany jest wyjątek `Program_Error`.

Przeterminowanie spotkań

```
task AlarmPozarowy is
  entry OK;
end AlarmPozarowy;

task body AlarmPozarowy is
begin
  NadzorPozarowy:
  loop
    select
      accept OK;           -- potwierdzenie brak alarmu
    or
      delay 5.0;
    ...                    -- podniesienie alarmu: OK timeout
    exit NadzorPozarowy;
  end select;
end loop NadzorPozarowy;
...                        -- operacje po ogłoszeniu alarmu
...                        -- np. rejestracja zdarzenia w dzienniku
end AlarmPozarowy;
```

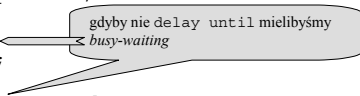
```
with Ada.Real_Time; use Ada.Real_Time;
-- importujemy Time, Clock, Time_span,
-- Miliseconds
task Nadzorca is
  entry Takt_nadzoru (Okres: in Time_Span);
end Nadzorca;

task body Nadzorca is
  Takt: Time_Span := Miliseconds(500);  -- okres
                                         -- początkowy
  Czas_odczytu: Time := Clock + Okres;
begin
  loop
    select
      accept Takt_nadzoru (Okres: in Time_Span) do
        Takt := Okres;
      end Takt_nadzoru;
    or
      delay until Czas_odczytu;
      Czas_odczytu := Czas_odczytu + Okres;
    end select;
  end loop;
end Nadzorca;
```

Gałąz else

Gałąz else pozwala serwerowi wycofać ofertę spotkania, gdy brak jest zadań-klientów już oczekujących na spotkanie. Gałąz else może wystąpić instrukcji select tylko raz i nie może być chroniona dozorem.

```
task body Nadzorca is
  Takt: Time_Span := Miliseconds(500);  -- okres
                                         -- początkowy
  Czas_odczytu: Time := Clock + Okres;
begin
  loop
    select
      accept Takt_nadzoru (Okres: in Time_Span) do
        Takt := Okres;
      end Takt_nadzoru;
    else
      null;
    end select;
    ...
    delay until Czas_odczytu;
    ...
    Czas_odczytu := Czas_odczytu + Okres;
  end loop;
end Nadzorca;
```



Gałąz terminate

Gałąz terminate jest wybierana gdy:

- jednostka macierzysta zadania zakończyła się i
- wszystkie zadania potomne albo zakończyły się, albo gotowe są wybrać gałąz terminate

Gałąz terminate może być poprzedzona dozorem. Nie może występować jednocześnie z gałęzią delay lub else.

```
task body Serwer is
begin
  loop
    select
      accept Entry1 (...) do ... end;
    or
      ...
    or
      terminate;
    end select;
  end loop;
end Serwer;
```

Terminowe wywołanie wejścia

Terminowe wywołanie wejścia jest anulowane, jeżeli nie zostało zaakceptowane w podanym czasie (względny lub bezwzględny).

```
task body SensorOgnioWy is
begin
  loop
    ...
    select
      Straznik.Alarm(...);
    or
      delay 2.0;
      StrazZawodowa.Pozar(...);
    end select;
  end loop;
end SensorOgnioWy;
```

Warunkowe wywołanie wejścia

Warunkowe wywołanie wejścia jest anulowane, jeżeli spotkanie nie może nastąpić natychmiast. Jest równoważne terminowemu wywołaniu wejścia z czasem przeterminowania o wartości zerowej.

```
task body SensorOgnioWy is
begin
  loop
    ...
    select
      Straznik.Alarm(...);
    else
      StrazZawodowa.Pozar(...);
    end select;
  end loop;
end SensorOgnioWy;
```

Literatura:

Konspekt wykładu:

<http://www.cs.put.poznan.pl/mszychowiak/dydaktyka/Wyklady/AdaSynchronizacja.ps.gz>

Inne:

[http://www.cs.put.poznan.pl/mszychowiak/dydaktyka/\[index.html#ADA\]](http://www.cs.put.poznan.pl/mszychowiak/dydaktyka/[index.html#ADA])

<http://unixlab.cs.put.poznan.pl/Ada95/>

Huzar, Fryźlewicz, Dubielewicz, Hnatkowska, Waniczek: „Ada95”, wyd. Helion 1998

<http://www.ci.pwr.wroc.pl/wzi/ada95.html>