Czas i przedział czasowy

- **Symulacja upływu czasu w komputerze:**
  - **Generator impulsów przesyłanych do licznika**
  - **Przepełnienie licznika powoduje zgłoszenie przerwania obsługiwanego przez system operacyjny**
  - **System w przerwaniu modyfikuje zegar programowy (przesuwa go o 1 tyknięcie)**
  - **Wartość tyknięcia zależy od dokładności symulacji pomiaru czasu i jest kompromisem pomiędzy precyzja a efektywnością.**
- **W klasycznych systemach operacyjnych:**
  - **System operacyjny aktualizuje zegar i kalendarz systemowy**
  - **Obiekty te są dostępne dla programów użytkowych przez wywołanie odpowiednich funkcji**
  - **W typowych aplikacjach dokładność odmierzania czasu nie jest zadaniem krytycznym**

Czas i przedział czasowy

- **W systemach czasu rzeczywistego oczekuje się, że:**
  - **Część zadań realizowanych jest cyklicznie z różnymi okresami aktywacji**
  - **Inne zadania muszą być aktywowane w określonych momentach czasowych**
- **Stąd na potrzeby systemów czasu rzeczywistego definiuje się:**
  - **Budziki (*watch dogs*), które nastawione na**
    - **określony moment czasowy (aktywacja w określonej chwili) lub**
    - **czas trwania (aktywacja po upływie pewnego czasu)**

    decydują o aktywacji przypisanych do nich procesów lub zadań.
- **Częstotliwość aktywacji niektórych zadań może być bardzo duża (systemy monitorowania i sterowania w czasie rzeczywistym procesów szybkozmiennych, np. monitorowanie drgań, sterowanie startem rakiety). W tych wypadkach wymagana jest duża dokładność pomiaru czasu, aby okres aktywacji zadań mógł być mierzony w milisekundach, a niekiedy dokładniej.**

Czas i przedział czasowy

- **Czynniki decydujące o dokładności określenia momentu wykonywania zadania:**
  - **Dokładność aktualizacji zegara**
    (w niektórych systemach można określić ilość impulsów generatora, co którą następuje przepełnienie licznika i zgłoszenie przerwania.)
  - **Faktyczny czas reakcji na przerwanie zegarowe**
    (Budzik przypisany do danego zadania przenosi je w stan gotowości (*ready*) ale nie powoduje jego uruchomienia. Np. jeśli priorytet zadania aktualnie wykonywanego jest wyższy od reaktywowanego zadania, to musi ono czekać na zakończenie obliczeń przez zadanie o wyższym priorytecie.

Ada – zestawienie mechanizmów czasowych

- **Pakiety umożliwiające dostep do zegara i datownika:**

  ```
  Ada.Calendar
  Ada.Real_Time
  ```

- **Definiowanie budzików możliwe jest dzięki instrukcjom:**

  ```
  delay
  delay until
  ```

Ada – pakiet *ada.calendar* (1)

```
package Ada.Calendar is

type Time is private;
subtype Year_Number  is Integer  range 1901..2099;
subtype Month_Number is Integer  range 1..12;
subtype Day_Number   is Integer  range 1..31;
subtype Day_Duration is Duration range 0.0..86_400.0;
function Clock return Time;
function Year    (Date : Time) return Year_Number;
function Month   (Date : Time) return Month_Number;
function Day     (Date : Time) return Day_Number;
function Seconds (Date : Time) return Day_Duration;
procedure Split  (  Date    : in Time;
                    Year    : out Year_Number;
                    Month   : out Month_Number;
                    Day     : out Day_Number;
                    Seconds : out Day_Duration);
function Time_of (  Year    : Year_Number;
                    Month   : Month_Number;
                    Day     : Day_Number;
                    Seconds : Day_Duration := 0.0)
                    return Time;
```

Ada – pakiet *ada.calendar* (2)

```
package Ada.Calendar is

function "+" (Left : Time;     Right : Duration) return Time;
function "+" (Left : Duration; Right : Time)     return Time;
function "-" (Left : Time;     Right : Duration) return Time;
function "-" (Left : Time;     Right : Time)     return
                                                    Duration;
function "<"  (Left, Right : Time) return Boolean;
function "<=" (Left, Right : Time) return Boolean;
function ">"  (Left, Right : Time) return Boolean;
function ">=" (Left, Right : Time) return Boolean;
TIME_ERROR : exception;

end Ada.Calendar;
```

Ada – pakiet *ada.real_time* (1)

```
package Ada.Real_Time is

type Time is private;
Time_First: constant Time;
Time_Last: constant Time;
Time_Unit: constant := 1.0/Integer(2000000000);
                                          -- Target dependent
type Time_Span is private;
Time_Span_First: constant Time_Span;
Time_Span_Last: constant Time_Span;
Time_Span_Zero: constant Time_Span;
Time_Span_Unit: constant Time_Span;

Tick: constant Time_Span;
function Clock return Time;
```

Ada – pakiet *ada.real_time* (2)

```
function "+"(Left: Time; Right: Time_Span) return Time;
function "+"(Left: Time_Span; Right: Time) return Time;
function "-"(Left: Time; Right: Time_Span) return Time;
function "-"(Left: Time; Right: Time) return Time_Span;

function "<"(Left, Right: Time) return Boolean;
function "<="(Left, Right: Time) return Boolean;
function ">"(Left, Right: Time) return Boolean;
function ">="(Left, Right: Time) return Boolean;

function "+"(Left, Right: Time_Span) return Time_Span;
function "-"(Left, Right: Time_Span) return Time_Span;
function "-"(Right: Time_Span) return Time_Span;
function "/"(Left: Time_Span; Right: Integer) return Time_Span;
function "/"(Left, Right: Time_Span) return Integer;
function "*"(Left: Time_Span; Right: Integer) return Time_Span;
function "*"(Left: Integer; Right: Time_Span) return Time_Span;
function "abs"(Right : Time_Span) return Time_Span;
function "<"(Left, Right: Time_Span) return Boolean;
function "<="(Left, Right: Time_Span) return Boolean;
function ">"(Left, Right: Time_Span) return Boolean;
function ">="(Left, Right: Time_Span) return Boolean;
```

Ada – pakiet *ada.real_time* (3)

```
function To_Duration(TS: Time_Span) return Duration;
function To_Time_Span(D: Duration) return Time_Span;
function Nanoseconds(NS: Integer) return Time_Span;
function Microseconds(US: Integer) return Time_Span;
function Milliseconds(MS: Integer) return Time_Span;
type Seconds_Count is range       -- Target dependent
      Integer'First .. Integer'Last;

procedure Split(    T : in Time; SC: out Seconds_Count;
                    TS: out Time_Span );
function Time_Of(SC: Seconds_Count; TS: Time_Span) return Time;

end Ada.Real_Time;
```

Ada – typ *duration* - uwagi

- **Typ *duration* jest typem stałoprzecinkowym, stąd:**
  - **stałe powinny mieć postać zawierającą kropkę dziesiętną,**
  - **dozwolone jest mnożenie i dzielenie przez liczbę całkowitą (*Integer*).**

- **Przykładowe poprawne deklaracje z zastosowaniem typu *duration*:**

```
Seconds: constant Duration :=0;
Minutes: constant Duration := 60.0*Seconds;
Hours:   constant Duration := 60.0*Minutes;
Period:  constant Duration := 2*Hours + 15*Minutes;
```

Ada – instrukcje *delay* i *delay until*

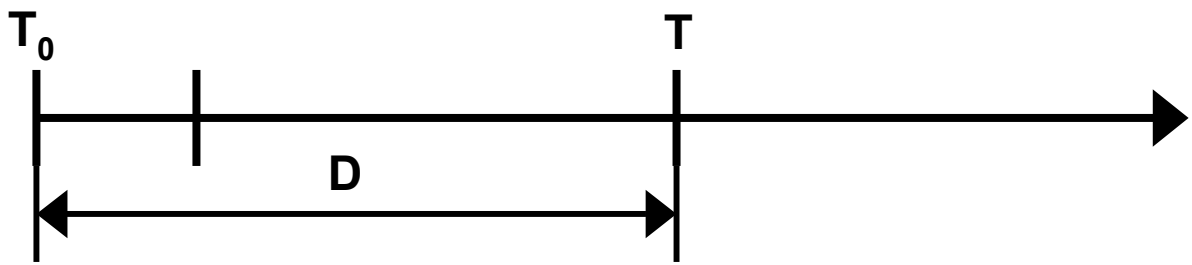
```
-- instrukcja delay:
delay Wyrażenie_Typu_Duration;

-- zawieś wykonującą jednostkę (zadanie) na czas równy
-- Wyrażenie_Typu_Duration

-- instrukcja delay until:
dealy until Wyrażenie_Typu_Time;

-- zawieś wykonującą jednostkę do momentu czasowego
-- obliczonego jako wartość Wyrażenia_Typu_Time
```

Uwaga:

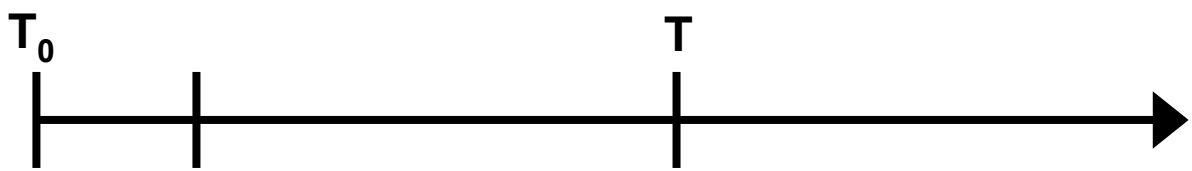
Zawieszenie zadania na określony czas lub do pewnego momentu czasowego nie oznacza, że zadanie to będzie wznowione i wykonywane po upływie tego czasu. Po upływie czasu zadanie przejdzie do fazy gotowe (*ready*), co nie oznacza przejścia do fazy wykonywania, gdyż w danym stanie systemu mogą się wykonywać inne zadania o wyższym priorytecie.

Ada – instrukcje *delay* i *delay until* - interpretacja

```
delay D;
```




```
dealy until T;
```



Ada – programowanie zadań cyklicznych

```
-- Zadanie cykliczne – pierwsza wersja
task type Semi_Periodic;

task body Semi_Periodic is
    Period: constant DURATION := 0.05;
begin
    loop        -- instrukcje wykonywane cyklicznie
                delay Period;
    end loop;
end Semi_Periodic;
```

Rozwiązanie nie zapewnia dokładnej cykliczności, nawet przy założeniu natychmiastowego wznowienia zadania po przejściu w fazę gotowe. Pojedyncza iteracja pętli, a w tym okres wykonywania instrukcji cyklicznych zawiera następujące składniki:

- **czas wykonywania instrukcji cyklicznych,**
- **opóźnienie 0,05 s specyfikowane przy instrukcji *delay***
- **czas wykonywania instrukcji organizujących pętli, a przynajmniej czas wykonania skoku bezwarunkowego.**

Okres aktywacji zadania nie wynosi 0,05, ale ma wartość większą, trudną do określenia na etapie pisania programu!

Ada – programowanie zadań cyklicznych

```
-- Zadanie cykliczne – eliminacja niekorzystnych składników
-- czasowych
task type Periodic;
   Seconds:   constant Duration := 1.0;
   Period:    constant Duration := 0.5 * Seconds;
   Offset:    constant Duration := 0.2 * Seconds;
   Next_Time : Time := Calendar.Clock + Offset;

task body Periodic is
begin
    loop
       delay until Next_Time;
       -- lub: delay Next_Time – Calendar.Clock;
       Put("Nastepny Tick"); New_Line(2);
       Next_Time := Next_Time + Period;
    end loop;
end Periodic;
```

Zadania w języku Ada – mechanizm synchronizacji

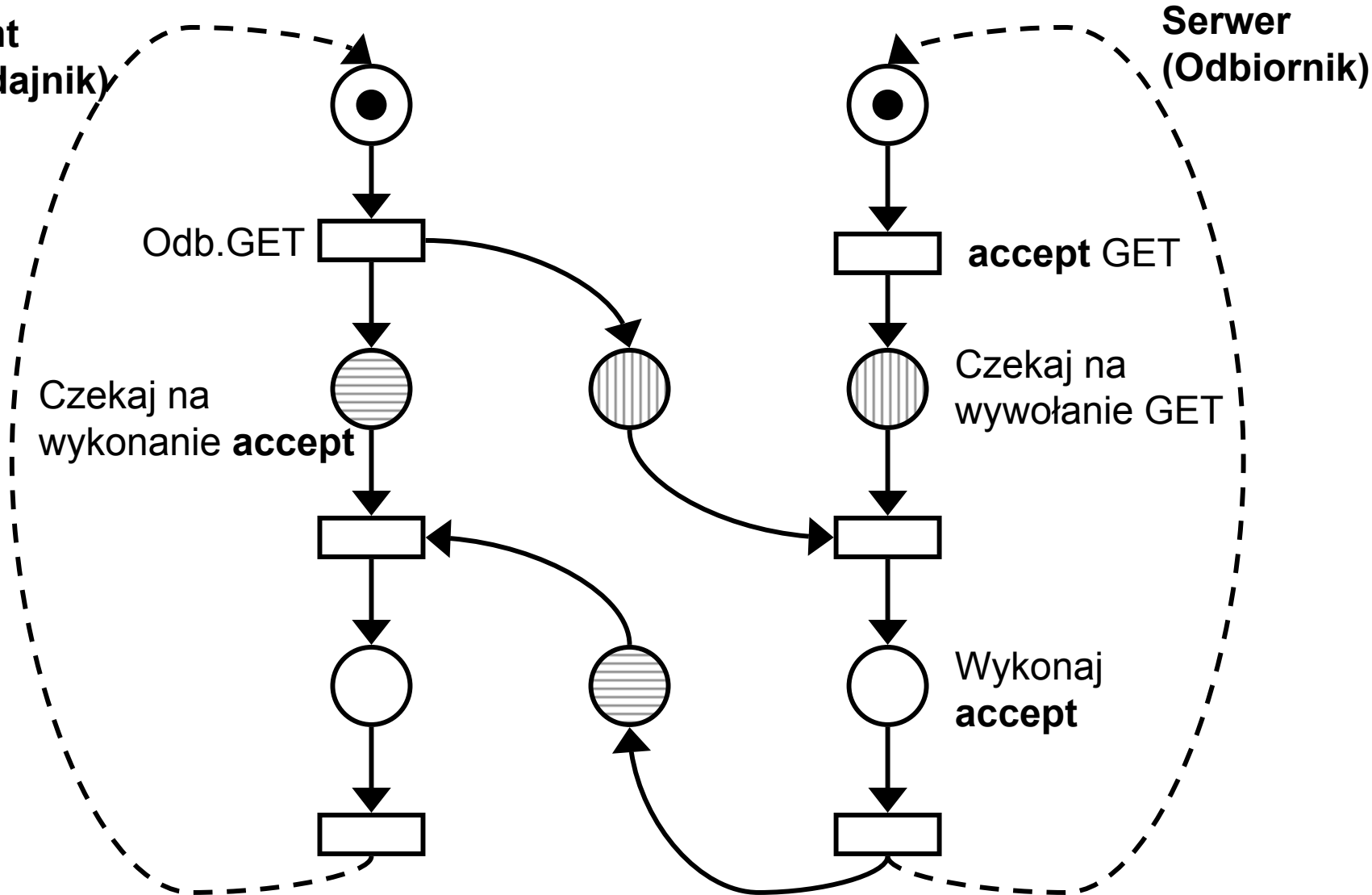
```
-- Instrukcja accept:
with Text_IO; use Text_IO;

procedure spotkanie is
    task type odbiornik is
        entry GET; -- zdefiniowanie wejścia
    end Odbiornik;
    task type Nadajnik;

    task body Odbiornik is
    begin
        loop   -- obsługa wejścia
               accept GET do
                      put("Odebrano komunikat");
                      New_Line;
               end GET;
        end loop;
    end Odbiornik;

    Odb : Odbiornik; -- uruchomienie zadania Odb
```

Zadania w języku Ada – mechanizm synchronizacji

```
-- Instrukcja accept (cd.):

    task body Nadajnik is
    c: Character;
    begin
        loop
                Get(c);
                exit when C='?';
                Odb.GET;        -- Konieczne było wcześniejsze
                                -- utworzenie zadania Odb.
        end loop;
    end Nadajnik;

    Nad : Nadajnik; -- uruchomienie zadania Nad
begin
    Put("Uruchomiono zadania");
    New_Line;
end Spotkanie;
```

Zadania w języku Ada – mechanizm synchronizacji

- **Mechanizm spotkania (*randezvous*):**



Zadania w języku Ada – przykład – semafor

```
with Text_IO; use Text_IO;
with ada.integer_text_io; use ada.integer_text_io;
with ada.float_text_io;   use ada.float_text_io;
with Ada.Calendar; use  Ada.Calendar;

procedure main is
-- Specyfikacja semafora
task type SEMAPHORE is
    entry ACCES_TO;
    entry FREE;
end SEMAPHORE;

-- Implmentacja semafora
task body SEMAPHORE is
begin
    loop
        accept ACCES_TO;
        accept FREE;
    end loop;
end SEMAPHORE;

Drukarka : Integer; -- zasób współdzielony
```

Zadania w języku Ada – przykład - semafor

```
Semafor : SEMAPHORE; -- uruchomienie semafora
-- zadanie z parametrami wejściowymi:
task type T(Init: Integer; Del: Integer);

task body T is
    dana : Integer;
begin
    loop
        Dana := Init;
        Semafor.ACCES_TO;    -- początek sekcji krytycznej
               Drukarka:=Dana;
               Put(Drukarka);
               New_Line;
        Semafor.FREE;        -- koniec sekcji krytycznej
        delay Duration(Del);
    end loop;
end T;

Task1: T(2,1);
Task2: T(4,2);

begin  null;end;
```

Ada – synchronizacja zadań z zastosowaniem pakietu Ada.Synchronous_Task_Control

```
-- predefiniowana implementacja binarnego semafora:

package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;

  procedure Suspend_Until_True(S : in out Suspension_Object);
        -- zgłasza Program_Error jeśli więcej niż jedno
        -- zadanie próbuje być zawieszone na jednym
        -- obiekcie zawieszającym
private
        -- nie wyspecyfikowane w języku
end Ada.Synchronous_Task_Control;
```

Ada - synchroniczne sterowanie zadaniami - przykład

```
-- szkic zastosowania pakietu Ada.Synchronous_Task_Control:
with Ada.Synchronous_Task_Control;
use Synchronous_Task_Control;
...
S_O: Suspension_Object;
...
task body T1 is
  begin loop
       -- zawieś zadanie dopóki wartość S_O nie wyniesie True:
            Suspend_Until_True(S_O);
            ...
       end loop;
  end;

  task body T2 is
  begin loop
            ...
       -- „odwieś” zadanie zablokowane na S_O:
            Set_True(S_O);
            ...
       end loop;
  end;
```

Modele komunikacji między procesami w programowaniu współbieżnym

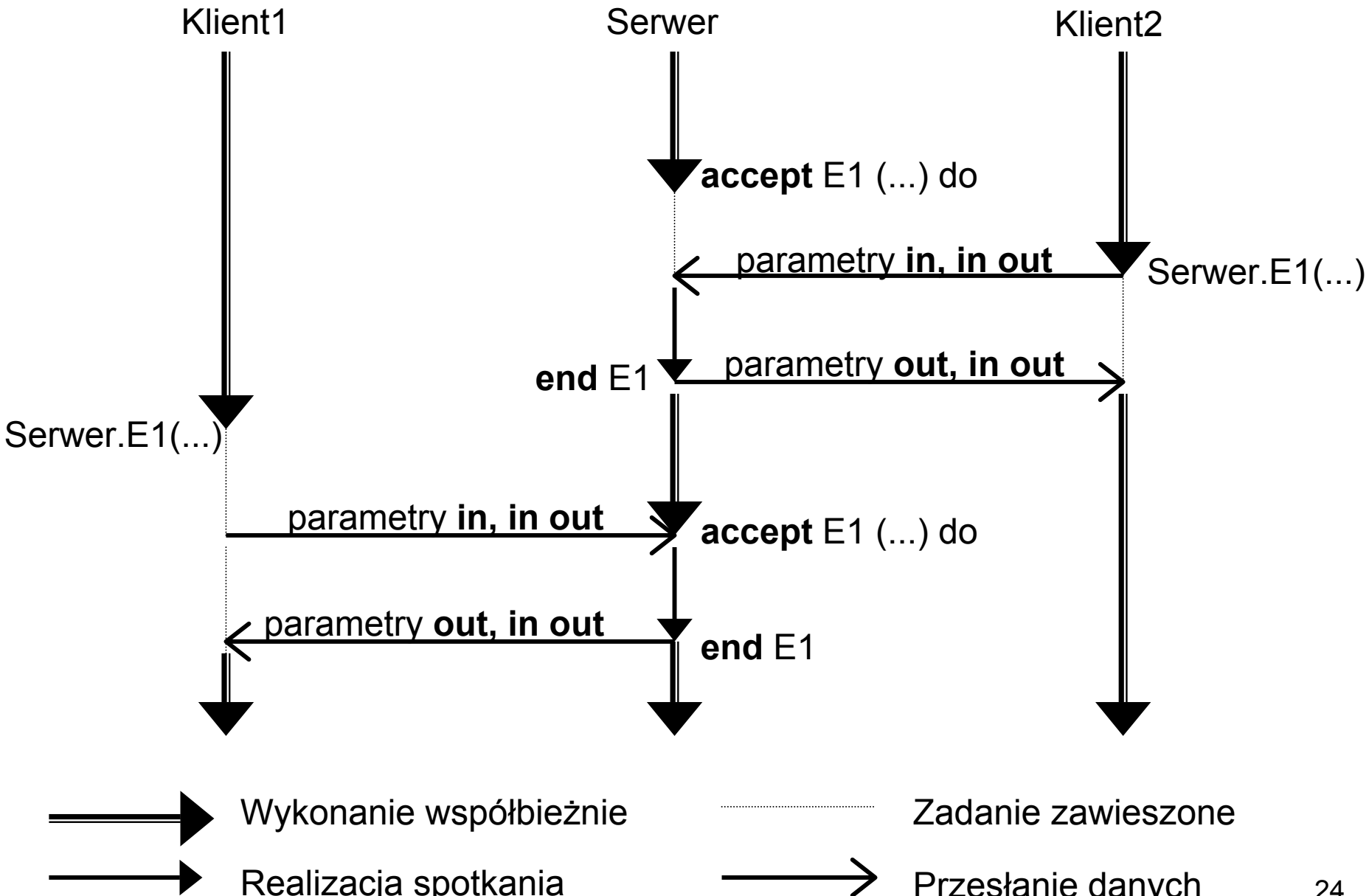
- **Wymiana komunikatów (*message passing*) – bezpośrednia wymiana danych pomiędzy procesami z zastosowaniem jakiegoś medium/ mechanizmu.**
- **Zmienne dzielone (*shared variables*)– obiekty, do których może mieć dostęp więcej niż jeden proces. Komunikacja może się odbywać przez odczyt/zapis tych zmiennych przez różne procesy, kedy jest to wymagane.**

Wybór metody komunikacji zależy od projektanta. Zmienne dzielone są łatwiejsze do implementacji, jeśli rzeczywiście istnieje współdzielony obszar pamięci pomiędzy procesami, ale może być zrealizowany nawet, jeśli struktura sprzętu zakłada jednostki obliczeniowe połączone tylko kanałami komunikacyjnymi. Podobnie wymiana komunikatów może może odbywać się zastosowaniem współdzielonej pamięci albo fizycznej sieci przekazywania komunikatów.

Sposoby synchronizacji podczas wymiany komunikatów

- **Brak synchronizacji – obiekt wysyłający pracuje dalej, po wysłaniu wiadomości, bez względu, czy wiadomość dotarła, czy nie (np. SO zgodne z POSIX)**
- **Synchronizacja – obiekt wysyłający pracuje dalej, tylko jeśli wiadomość dotarła.**
- **Zdalne wywołanie – obiekt wysyłający kontynuuje obliczenia tylko, jeśli otrzyma odpowiedź od adresata wiadomości (np. ADA).**

Ada – komunikacja w trakcie spotkania



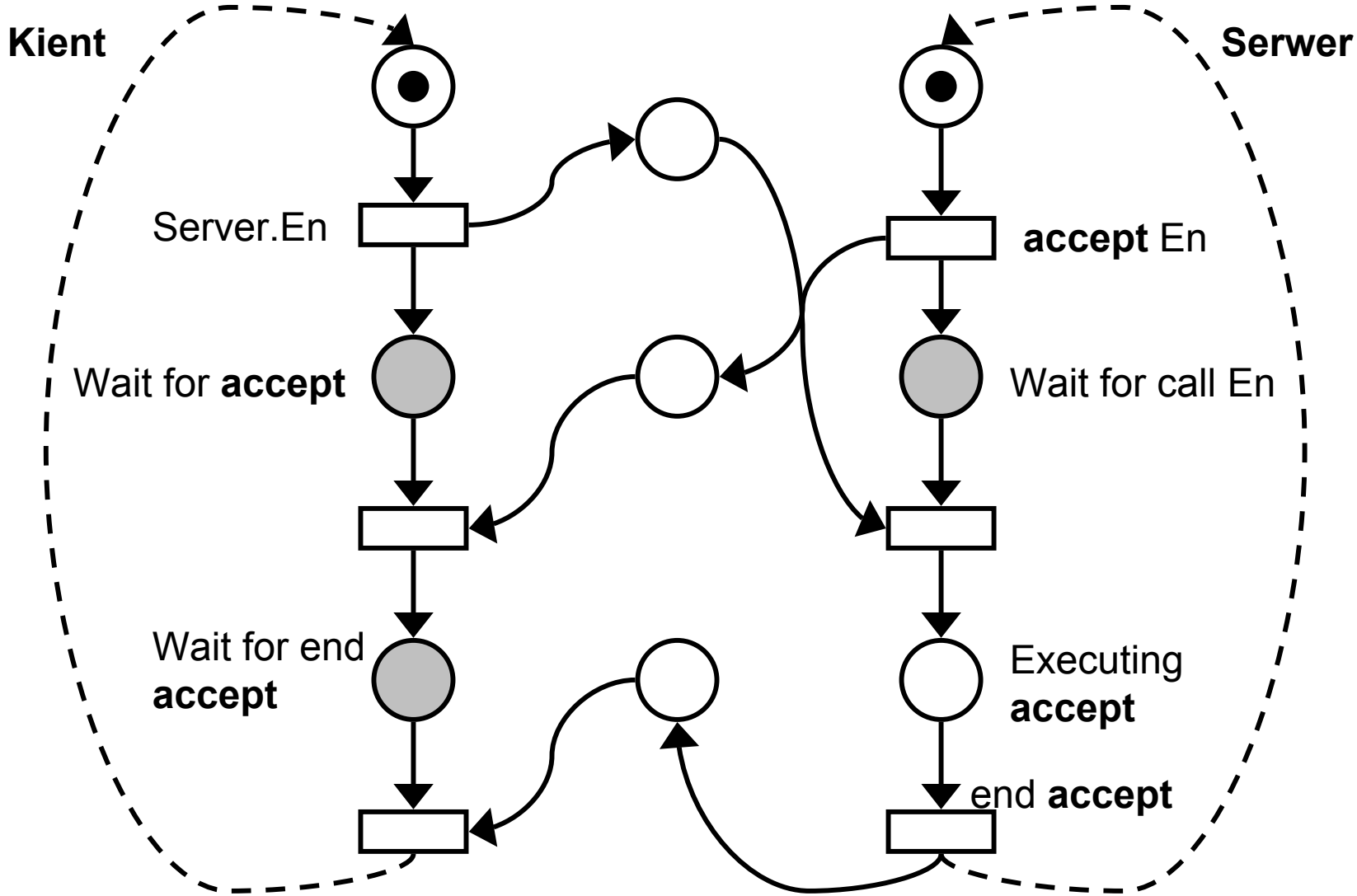
Ada – komunikacja w czasie spotkania – przykład wstępny (1)

```
with Text_IO; use Text_IO;
with ada.integer_text_io; use ada.integer_text_io;
with ada.float_text_io;   use ada.float_text_io;

procedure spotkanie is
    task type odbiornik is
        entry GET (C: in Character);
    end Odbiornik;

    task body Odbiornik is
    Znak : Character;
    begin
        loop
                accept GET(C: in Character) do
                        Znak := C;
                        put("Odebrano komunikat: ");Put(Znak);
                        New_Line;
                end GET;
        end loop;
    end Odbiornik;
    Odb : Odbiornik; -- uruchomienie zadania
```

Ada – komunikacja w czasie spotkania – przykład wstępny (2)

```
    task type Nadajnik;
    task body Nadajnik is
    c: Character;
    begin
        loop
                Get(c);
                exit when C='?';
                Odb.GET(C);
        end loop;
    end Nadajnik;

    Nad : Nadajnik; -- uruchomienie zadania
begin
    Put("Uruchomiono zadania");
    New_Line;
end Spotkanie;
```

Ada – synchronizacja mechanizmu spotkania



Ada – synchronizacja mechanizmu spotkania - uwagi

- Wiele klientów może wywoływać dane serwera
- Żądania wołania ustawiane są w kolejce do wejścia
- Klienci oczekują wtedy na osiągnięcie swojego punktu synchronizacji
- Kolejki przy wejściu mogą być organizowane według różnych strategii (domyślnie przyjmowany jest reżim FIFO)
- Ilość żądań oczekujących w kolejce do danego wejścia jest określana przez atrybut `Count` (np. dla wejścia En wartosc tego atrybutu może być otrzymana jako wartość wyrażenia `En'Count`).

Ada –wywołanie wejścia a wywołanie procedury - uwagi

- **Wywołanie wejścia nie jest tożsame z wywołaniem procedury ponieważ:**
  - **Wykonanie obsługi wołanego wejścia może być opóźnione do czasu osiągnięcia przez serwer odpowiedniego punktu synchronizacji;**
  - **W trakcie spotkania instrukcje obsługi wejścia (wnętrze odpowiedniej instrukcji accept) są wykonywane w trybie wzajemnego wykluczania (wielokrotne wywołania są realizowane po kolei, podczas gdy procedury mogą być wywoływane współbieżnie przez procedury);**
  - **Zadanie wołane (serwer) może zadecydować o możliwości wykonania obsługi danego wołania przez uzależnienie od stanu.**

Ada – wykonanie obsługi danego wołania w zależności od stanu

```
task body SERVICE is
    Switch: Boolean;
Begin
    -- ...
    if Switch then -- w zależności od zmienne Switch dopuszcza
                   -- się wykonywanie wejść E1 lub E2
        accept E1 do
               --...
        end E1;
    else
        accept E2 do
               --...
        end E2;
    end if;
    -- ...
end SERVICE;
```

Ada – synchronizacja przez sekwencję zapisu

```
task type Switcher is
    entry Press_Button;
end Switcher;

task body Switcher is
begin
    loop
        accept Press_Button;
        Switch_Light_On;
        accept Press_Button;
        Switch_Light_Off;
    end loop;
end Switcher;
```

Ada – komunikacja w trakcie spotkania

- **Składnia parametrów przesyłanych w wyniku wywoływania i zakończenia wejścia jest identyczna jak dla wywołania i powrotu procedury;**
- **Parametry typu wejściowego (*in, in out*) są przesyłane z zadania wołającego (klient) do zadania wołanego (serwer) oraz odpowiednio, parametry wyjściowe (*out, in out*) – z przestrzeni serwera do klienta.**
- **Wartość parametrów przesyłanych w trybie *in* obliczana jest bezpośrednio przed wywołaniem danego wejścia, a nie w punkcie synchronizacji (spotkania). Może zaistnieć taka sytuacja, że wartości przesyłanych argumentów nie będą odpowiadać aktualnym wartościom zmiennych, np.:**

```
if Gloobal_Variable > 20 then T.E;
-- w zadaniu wołającym (klient)
...
-- zadanie T (wołane)
accept E do
    ... Global_Variable
    -- w zadaniu wołanym (serwer)
end E;
```

Zadania w języku Ada – przykład producent - konsument

```
-- specyfikacja pakietu z zadaniem:

package buffer1 is
       task type Buf1 is
              entry Put(X: in Integer);
              entry Get(X: out Integer);
       end Buf1;
end Buffer1;
```

Zadania w języku Ada – przykład producent - konsument

```
-- implementacja pakietu z zadaniem:

package body Buffer1 is
       task body Buf1 is
              V: Integer;
       begin
              loop
                     accept Put(x: in Integer) do
                            V:=X;
                     end Put;
                     accept Get(X: out Integer) do
                     X:=V;
                     end Get;
              end loop;
       end Buf1;
end Buffer1;
```

Zadania w języku Ada – przykład producent - konsument

```
-- Program producent – bufor - konsument:
with ada.integer_text_io; use ada.integer_text_io;
ith Buffer1; use Buffer1;
procedure main is
    task producer;
    task consumer;
    Buffer: Buf1;
    task body Producer is
    begin      for I in 1..10 loop
               Buffer.Put(Integer(i));
               end loop;
    end Producer;
    task Body Consumer is
       Dana : Integer;
    begin  loop
               Buffer.Get(Dana);
               Put(Dana); New_Line;
           end loop;
    end Consumer;

begin  null; end Main;
```

Obiekty chronione komunikacja przez zmienne dzielone

- **Obiekty chronione stanowią konstrukcję zamykająca wewnątrz dane (o dowolnej strukturze), które są dostępne z zewnątrz przez udostępnianie operacji – podprogramów lub wejść.**
- **Wywołanie operacji zewnętrznych obiektu chronionego jest synchronizowane w celu zapewnienia poprawności (wzajemne wykluczanie) współbieżnego przetwarzanie danych wewnątrz obiektu.**
- **Obiekt chroniony odpowiada monitorowi.**
- **Obiekt chroniony w odróżnieniu od zadania jest wykonywany wyłącznie, jeśli wywołana zostanie pewna jego operacja, po zakończeniu której obiekt jest gotowy do realizacji następnej operacji Nie występuje oddzielny własny wątek wykonywania obiektu; synchronizacja dotyczy jedynie zapewnienia wzajemnego wykluczania odpowiednich operacji.**

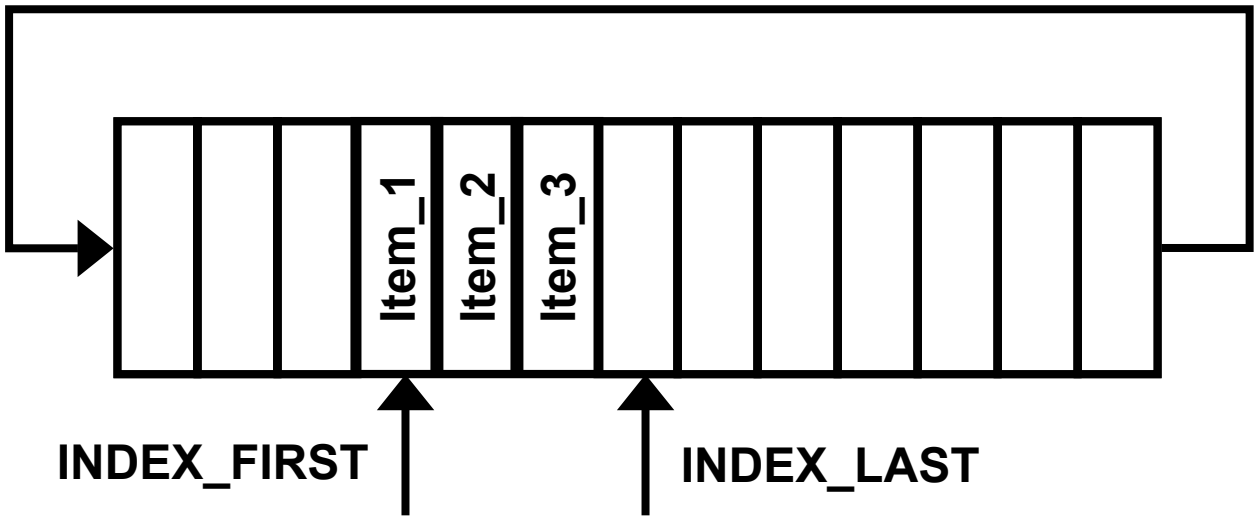
Semantyka obiektu chronionego

- **Typ chroniony zamyka we wnętrzu dane, które są dostępne z zewnątrz przez wywołanie specyfikowanych w interfejsie funkcji, procedur lub wejść. Spełnione są przy tym następujące warunki:**
  - **Możliwe jest współbieżne wykonywanie wielu wywołanych funkcji obiektu chronionego. Funkcja ma możliwość jedynie czytania obiektu chronionego. Wykonywanie funkcji jest realizowane na zasadzie wyłączności względem procedur i wejść.**
  - **Procedury obiektu chronionego mogą być wykonywane na zasadzie wyłączności, tzn. jeśli wykonywana jest procedura obiektu chronionego, to jest ona jedyną jednostką wykonywaną w danym momencie.**
  - **Wykonanie wołanego wejścia możliwe jest na zasadzie wyłączności względem innych elementów (podobnie jak procedury). Możliwość wykonywania wejścia jest dodatkowo ograniczana przez wyrażenie logiczne („Warunek Bariery"). Oznacza to, że oprócz warunków synchronizacji, konieczne jest również spełnienie tego wyrażenia logicznego. Odpowiada to konstrukcji warunkowego obszaru krytycznego.**

Dana dzielona implementowana w obiekcie chronionym

```
protected type Shared_Data(Initial_Value: Integer) is
   function Read return Integer;
   procedure Write(New_Value: Integer);
   -- tu mogą być deklarowane również wejścia jak w zadaniach!
private
   -- deklaracje danych dzielonych niedostępnych na zewnątrz:
   Data: Integer := Initial_Value;
end Shared_Data;
My_Data : Shared_Data(125);

protected body Shared_Data is
   function Read return Integer is -- czytanie może odbywać się
   begin                            -- współbieżnie z innym
        return Data;                -- czytaniem
   end Read;
   procedure Write(New_Value: Integer) is
   begin
        Data:=New_Value;
   end Write;
end Shared_Data;
```

Bufor cykliczny – zasada działania:



Założenia implementacyjne:

- **Bufor ma ograniczona pojemność (SIZE),**
- **Elementy bufora będą typu całkowitego (Integer),**
- **Dostęp do bufora będzie realizowany przez wejścia WRITE i READ,**
- **Zastosowane będą dwa wskaźniki**
  - **INDEX_LAST – pierwsze wolne miejsce**
  - **INDEX_FIRST – pierwszy element w buforze**

Bufor cykliczny implementowany przez obiekt chroniony

```
Size : constant Natural := 100;
type Index is mod Size;
subtype Counter is Natural range 1..Size;
type Buffer is Array(Index) of Integer;

protected  type Bounded_Buffer is
   entry Read(Data: out Integer);
   entry Write(Data: in Integer);
private
   Index_First: Index := Index'First;
   Index_Last: Index := Index'Last;
   Counter: Natural range 0..Size :=0;
   Buffer_Mem: Buffer;
end Bounded_Buffer;
```

Bufor cykliczny implementowany przez obiekt chroniony (cd.)

```
protected body Bounded_Buffer is
   -- wejście chronione:
   entry Read(Data: Out Integer) when Counter /=0 is
   begin
         Data:=Buffer_Mem(Index_First);
         Index_First:= Index_First + 1;
         Counter := Counter - 1;
   end Read;

   entry Write(Data: in Integer) when Counter /=Size is
   begin
        Index_Last := Index_Last + 1;
        Buffer_Mem  (Index_Last) := Data;
        Counter := Counter + 1;
   end Write;
end Bounded_Buffer;
```

Bufor cykliczny – przykładowa aplikacja (1)

```
with Ada.Text_IO; use Ada.Text_IO;
with ada.integer_text_io; use ada.integer_text_io;
with Ada.Float_Text_IO;  use Ada.Float_Text_IO;
with Calendar; use Calendar;

procedure main is
   type moja_dana is mod 10;
   Dana : Moja_Dana := 0;
   Size : constant Natural := 100;
   type Index is mod Size;
   subtype Counter is Natural range 1..Size;
   type buffer is Array(Index) of Integer;

   protected  type Bounded_Buffer is
        entry Read(Data: out Integer);
        entry Write(Data: in Integer);
   private
        Index_First: Index := Index'First;
        Index_Last: Index := Index'Last;
        Counter: Natural range 0..Size :=0;
        Buffer_Mem: Buffer;
   end Bounded_Buffer;
```

Bufor cykliczny – przykładowa aplikacja (2)

```
protected body Bounded_Buffer is
   entry Read(Data: Out Integer) when Counter /=0 is
   begin
        Data:=Buffer_Mem(Index_First);
        Index_First:= Index_First + 1;
        Counter := Counter - 1;
   end Read;

   entry Write(Data: in Integer) when Counter /=Size is
   begin
        Index_Last := Index_Last + 1;
        Buffer_Mem  (Index_Last) := Data;
        Counter := Counter + 1;
   end Write;
end Bounded_Buffer;

My_Bounded_Buffer : Bounded_Buffer;
```

Bufor cykliczny – przykładowa aplikacja (3)

```
procedure Computes(Y: in Integer) is
Begin  put("Konsumuje wartosc"); put(Y); New_Line;
       Delay 0.4;
End;

procedure Produces(E: out Integer) is
Begin
  Dana:= Dana+1;    Delay 1.0;    E:=Integer(Dana);
End Produces;
```

Bufor cykliczny – przykładowa aplikacja (4)

```
task consumer is entry provides(x: in Integer);
end consumer;
task producer;

task body consumer is
  Y: Integer;
begin
  loop        My_Bounded_Buffer.Read(Y);        computes(y);
  end loop;
end consumer;

task body producer is
  E: Integer;
begin
  loop
      produces(E);        My_Bounded_Buffer.Write(E);
  end loop;
end producer;


begin  null; end main;
```