Helion

Paweł Majdzik

Programowanie współbieżne

Systemy czasu rzeczywistego

Współbieżność to szybkość, efektywność i nowoczesność.
Czy Ty też chcesz tak programować?

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Michał Mrowiec
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?prowsp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-4302-8

Copyright © Helion 2012

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Rozdział 8.

Problemy programowania współbieżnego

Do klasycznych problemów programowania współbieżnego należą problem producenta i konsumenta, problem pisarzy i czytelników oraz problem pięciu filozofów. W poprzednich rozdziałach, przy okazji prezentacji metod weryfikacji poprawności programów współbieżnych oraz opisu różnych mechanizmów synchronizacji, część tych problemów została mniej lub bardziej szczegółowo omówiona. Celem tego rozdziału jest nie tylko szczegółowe przedstawienie wymagań dotyczących zasad współpracy procesów w klasycznych problemach współbieżnych, ale przede wszystkim przedstawienie różnych rozwiązań (przy zastosowaniu różnych mechanizmów synchronizacji) oraz ocena ich efektywności.

Ten rozdział stanowi pewne podsumowanie dotychczas prezentowanych rozwiązań problemów programowania współbieżnego oraz przede wszystkim mechanizmów synchronizacji, zarówno tych dostępnych w Adzie (obiekty chronione i spotkania), jak i klasycznych mechanizmów synchronizacji (monitory, semafory)[1]. Dlatego też w tym miejscu pozwolono sobie na przypomnienie struktury niektórych już zaprezentowanych rozwiązań i ich uproszczoną implementację, jednak skupiono szczególną uwagę na tych rozwiązaniach, które będą prezentowane po raz pierwszy.

8.1. Problem konsumenta i producenta

Problem producenta i konsumenta jest abstrakcją wielu rzeczywistych problemów występujących w systemach operacyjnych i w szczególności w wielomarszrutowych systemach produkcyjnych. Przykładem może być współpraca procesów w systemach

[1] Czytelnik zapewne zauważył, że choć obiekt chroniony jest jednym z dwóch podstawowym mechanizmów synchronizacji zadań w Adzie, to w poprzednich rozdziałach w sposób bardziej szczegółowy omówiono instrukcję rekolejkowania i mechanizm asynchronicznej zamiany sterowania zadaniem. W tym rozdziale jest miejsce na wiele przykładów rozwiązań problemów współbieżnych opartych na obiekcie chronionym.

operacyjnych, w których programy użytkowe dokonują analizy i przetwarzania pewnych zestawień danych, które z kolei muszą być zinterpretowane przez inne procesy użytkownika lub procesy systemu operacyjnego, m.in. prosty program obsługi danych z urządzenia wejściowego (np. klawiatura), który jest konsumowany przez program użytkowy.

W problemie producenta i konsumenta producent cyklicznie produkuje porcję danych i przekazuje ją konsumentowi, który konsumuje ją w swojej sekcji lokalnej. Struktura procesów konsumenta i producenta jest następująca:

```
task Producent;
task body Producent is
  info: Integer := 1;
begin
  loop
       Produkuj;                        -- sekcja lokalna
    Protokół_wstępny;                   -- sprawdza, czy są wolne miejsca
      Umieszcza_dane_w_buforze;         -- lub przekazuje bezpośrednio konsumentowi
   Protokół_końcowy;                    -- sygnalizuje umieszczenie danych w buforze
  end loop;
end Producent;
task Konsument;
task body Konsument is
  info: Integer := 1;
begin
  loop
    Protokół_wstępny;                   -- sprawdza, czy są dane w buforze
       Pobiera_dane_z_bufora;           -- lub otrzymuje bezpośrednio od producenta
    Protokół_końcowy;                   -- sygnalizuje o zwolnieniu miejsca w buforze
      Konsumuj;                         -- sekcja lokalna
  end loop;
end Konsument;
```

W literaturze rozważa się implementacje dla różnych rozmiarów bufora:

- bufor jednostkowy (o pojemności równej 1);
- bufor nieskończony;
- bufor ograniczony (n-elementowy).

Najprostszy przypadek problemu producenta i konsumenta sprowadza się do synchronizowania dwóch procesów: producenta i konsumenta. Producent cyklicznie produkuje jedną porcję danych i przekazuje ją bezpośrednio konsumentowi. W przypadku komunikacji synchronicznej porcja danych będzie przekazana, gdy producent jest gotów do wysłania danych, a konsument do ich odebrania. Jeśli jeden z procesów nie jest gotowy do wysłania lub pobrania danych, to zostaje zawieszony w oczekiwaniu na drugi, co oznacza, że reprezentacja bufora w tym rozwiązaniu jest zbędna.

Większą elastyczność wykonywania procesów zapewnia komunikacja asynchroniczna pomiędzy producentem a konsumentem, uzyskiwana poprzez wprowadzenie bufora. W tym przypadku producent po wyprodukowaniu danych umieszcza je w buforze i może wykonywać kolejne operacje bez względu na to, czy konsument w danej chwili może pobrać porcje, czy też nie.

Bufor stanowi listę zawierającą dane wyprodukowane przez producenta, który umieszcza dane na końcu listy (dokładnie w pierwszym wolnym miejscu w buforze), konsument natomiast pobiera dane z początku listy (dokładnie z pierwszego miejsca, w którym zostały umieszczone dane). Takie rozwiązanie zapewnia, że porcje będą konsumowane w kolejności ich wyprodukowania, co jest jednym z założeń dotyczących współpracy procesów w rozważanym problemie. Producent może w dowolnej chwili umieścić porcje danych w niepełnym buforze, konsument natomiast w dowolnej chwili może pobrać dane z niepustego bufora.

Bufor nieskończony nie ma praktycznego zastosowania i jego implementacja może być wstępem dla poprawnej implementacji buforów skończonych[2].

W rzeczywistych systemach mamy do czynienia najczęściej z buforem ograniczonym i to on będzie podmiotem prezentowanych w tym rozdziale rozwiązań. Ciągłość wykonywania procesów producenta i konsumenta zależy od rozmiaru bufora. Z kolei wyznaczenie odpowiedniego rozmiaru bufora zależy od względnej liczby producentów i konsumentów oraz od względnej prędkości wykonania procedur produkcji i konsumpcji. Zauważmy, że jeżeli producenci szybciej produkują, niż konsumenci konsumują dane, to dowolny rozmiar bufora okaże się po pewnym czasie za mały, a w sytuacji odwrotnej, kiedy to konsumenci szybciej konsumują, niż producenci produkują, dowolny rozmiar bufora okaże się nadmiarowy. Łatwo zauważyć, że większy rozmiar bufora minimalizuje (lub w szczególnym przypadku likwiduje) czas oczekiwania producentów na wolne miejsca w buforze. Jednak w rzeczywistych systemach rozmiar bufora ma wpływ na koszty budowy systemu, stąd problem określenia satysfakcjonującego rozmiaru bufora jest jednym z głównych problemów stojących przed projektantami Elastycznych Systemów Produkcyjnych — w tych systemach bufory mogą reprezentować magazyny lub liczbę podajników w danym gnieździe albo stacji montażowej[3].

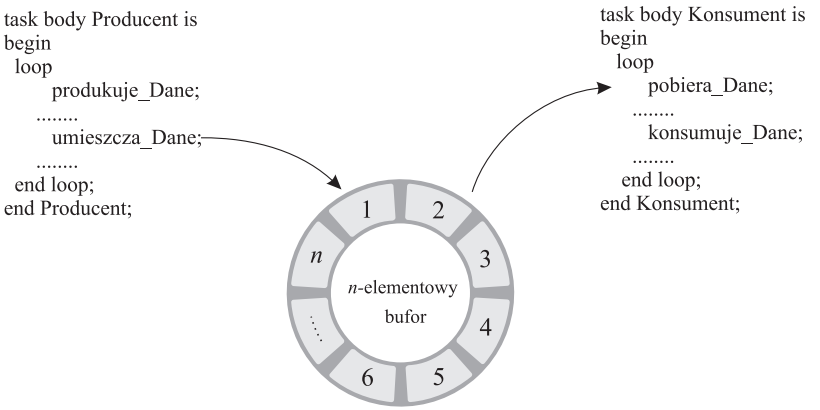
Implementacje bufora jednoelementowego oraz nieskończonego są szczególnymi przypadkami implementacji *n*-elementowego bufora i dlatego nie będą przedmiotem rozważań. Reprezentacją w programie *n*-elementowego bufora może być bufor cykliczny (rysunek 8.1). Jest on prosty i wygodny w implementacji, ale ma zastosowanie jedynie w systemach, w których rozmiar bufora jest stały. Implementacja problemu producenta i konsumenta z buforem o zmiennym rozmiarze lub z pulą buforów są przedmiotem zadań do wykonania dla Czytelników.

Ponadto oprócz klasycznego wariantu, w którym występują tylko dwa procesy — jeden proces producenta oraz jeden proces konsumenta — często problem dotyczy synchronizacji wielu producentów i wielu konsumentów. Należy podkreślić, że poprawność prezentowanych rozwiązań nie zależy od względnej liczby producentów i konsumentów oraz od prędkości wykonywania.

---

[2] Łatwo zauważyć, że w przypadku implementacji bufora nieskończonego programiści muszą jedynie skupić się na synchronizacji procesów konsumenta, tak aby nie pobierały one porcji z pustego bufora. Te reguły synchronizacji mogą być bezpośrednio przeniesione do implementacji bufora nieskończonego, rozszerzone o reguły synchronizacji producentów.

[3] W przypadku ogólnym problem określenia optymalnego rozmiaru bufora należy do klasy problemów NP-zupełnych.



Rysunek 8.1. *Bufor cykliczny*

Efektywne rozwiązanie problemu producenta i konsumenta umożliwia jednoczesne pobieranie i wstawianie danych — odpowiednio — przez konsumenta i producenta do różnych elementów ograniczonego bufora. Takie przetwarzanie danych zwiększa stopień rzeczywistego, równoległego wykonywania procesów. Ten stopień równoległości wykonywania procesów będzie jednym z podstawowych kryteriów prezentowanych rozwiązań.

8.1.1. Semafory

Na początku rozważmy najbardziej klasyczny przypadek problemu producenta i konsumenta z reprezentacją *n*-elementowego bufora. W poniższym przykładzie zastosowano bufor cykliczny, zatem indeks elementu bufora, do którego producent może wstawić lub z którego konsument może pobrać dane, jest obliczany jako modulo rozmiaru bufora. Należy zauważyć, że procesy producenta i konsumenta muszą mieć własny wskaźnik określający — odpowiednio — pierwsze wolne miejsce w buforze i miejsce najwcześniej umieszczonych danych[4].

```
with Semafory; use Semafory;
procedure ProducentKonsument is
  task Producent;
  task Konsument;
  n: constant Integer := 10;        -- rozmiar bufora
  Miejsca, Dane: Sem;               -- wartość początkowa semaforów, odpowiednio, równa n i 0
  Bufor: array(1..n) of Integer;

  task body Producent is
  wskP: Integer := 0;
  info: Integer := 0;
    begin
      loop
          produkuje_Dane;
          Miejsca.wait;
          wskP := (wskP mod n) + 1;
```

[4] We wszystkich rozwiązaniach opartych na semaforach są stosowane typy `Sem`, `SemBin` oraz `Sem_ograniczony` zdefiniowane w pakiecie `Semafory` opisanym w podrozdziale 6.5.

```
          Bufor(wskP) := info;
        Dane.signal;
      end loop;
  end Producent;
  task body Konsument is
    wskK: Integer := 0;
    info: Integer := 0;
  begin
    loop
      Dane.wait;
        wskK := (wskK mod n) + 1;
        info := Bufor(wskK);
      Miejsca.signal;
        konsumuje_Dane;
    end loop;
  end Konsument;
begin
  Miejsca.init(n);                -- ustawienie wartości początkowej semaforów
  Dane.init(0);
end ProducentKonsument;
```

Wartość semafora ogólnego `Miejsca`, o wartości początkowej równej rozmiarowi bufora, określa liczbę wolnych miejsc w buforze, a wartość semafora `Dane`, o wartości początkowej równej 0, określa liczbę danych w buforze. Operacja `Miejsca.wait` (stanowi protokół wstępny) jest wykonywana przez producenta przed wstawieniem porcji danych do bufora, operacja `Dane.signal` (stanowi protokół końcowy) jest natomiast wykonywana przez producenta po umieszczeniu porcji danych i ma na celu powiadomienie konsumenta o nowej porcji danych w buforze. Analogicznie operacja `Dane.wait` jest wykonywana przez konsumenta przed pobraniem porcji danych z bufora i jest możliwa do wykonania, jeśli w buforze są jakiekolwiek dane. Operacja `Miejsca.signal` jest natomiast wykonywana przez konsumenta po pobraniu danych z bufora i powiadamia producenta o zwolnieniu miejsca w buforze.

Jeśli wartość semafora `Miejsca` jest równa 0, to producent umieścił kolejno `n` porcji danych, bez przeplotu operacji pobierania danych przez konsumenta. W tym stanie bufora, jeżeli producent chce umieścić kolejną porcję danych, zostaje zawieszony na semaforze `Miejsca` do czasu, aż proces konsumenta pobierze porcję danych i tym samym zwolni miejsce w buforze (konsument sygnalizuje to wykonaniem operacji `Miejsca.signal`). Z kolei wartość semafora `Dane` równa 0 oznacza, że nie ma porcji danych w buforze i proces konsumenta, który chce pobrać dane, jest zawieszony na tym semaforze do czasu, aż w buforze pojawi się nowa porcja danych (producent sygnalizuje to wykonaniem operacji `Dane.signal`).

Należy jeszcze wykazać, że w tym samym czasie producenci i konsumenci nie będą odpowiednio wstawiać i pobierać danych z tego samego miejsca w buforze. Zauważmy, że po każdym umieszczeniu lub pobraniu porcji danych z bufora (procesy wykonują swoje sekcje lokalne) suma wartości semafora `Miejsca` i `Dane` jest zawsze równa `n`. Każdemu opuszczeniu semafora `Miejsca` (zmniejszenie o 1) przed umieszczeniem porcji danych towarzyszy podniesienie semafora `Dane` po umieszczeniu porcji (zwiększenie o 1). Analogiczna sytuacja ma miejsce, gdy konsument pobiera porcję danych, wykonując `Dane.wait`, i sygnalizuje o wolnym miejscu wykonaniem operacji `Miejsca.signal`.

Ponadto wartości zmiennych wskP i wskK są równe (tzn. wskaźniki wskP i wskK wskazują to samo miejsce w buforze) tylko wtedy, gdy bufor jest pusty lub pełny. Oznacza to, że gdy wskP = wskK, to jeden z pary semaforów Miejsca lub Dane ma wartość 0, co jednocześnie uniemożliwia wykonanie dwóch operacji: pobrania danych przez konsumenta i wstawienia danych przez producenta. W przypadku gdy wartości semaforów Miejsca i Dane są różne od zera, jest zapewnione, że jednoczesne pobieranie i umieszczanie porcji danych dotyczy różnych elementów bufora.

Implementacja bufora dla wielu producentów i konsumentów wymaga deklaracji zmiennych globalnych wskP i wskK, ponieważ te zmienne muszą być współdzielone przez producentów i konsumentów. Jednak samo przekształcenie zmiennych lokalnych wskP i wskK na zmienne globalne nie gwarantuje poprawnej synchronizacji procesów w dostępie do bufora.

Rozważmy stan programu, w którym bufor ma co najmniej dwa miejsca wolne oraz procesy Producent1 i Producent2 chcą w tym samym czasie umieścić w nim porcje danych. Każdy z producentów może wykonać operację opuszczania semafora Miejsca.wait, ponieważ wartość semafora Miejsca jest równa co najmniej 2. Wówczas możliwy jest następujący przeplot operacji (założono, że wskP = 1, zatem wolny jest drugi i trzeci element bufora):

- Producent1 wykonuje operację określenia wolnego miejsca w buforze — wskP := (wskP mod n) + 1, więc wskP := 2;
- Producent2 wykonuje operację określenia wolnego miejsca w buforze wskP := (wskP mod n) + 1, więc wskP := 3;
- Producent2 wstawia dane do trzeciego elementu bufora, Bufor(wskP) := info;
- Producent1 wstawia dane do trzeciego elementu bufora, Bufor(wskP) := info.

Po wykonaniu powyższego przeplotu operacji jeden z konsumentów będzie pobierać dane z drugiego, pustego elementu bufora. Ponadto dane wyprodukowane przez pierwszego producenta zostały nieskonsumowane i nadpisane danymi wygenerowanymi przez drugiego producenta. W przypadku gdy umieszczane w buforze dane stanowią stosunkowo dużą strukturę, to istnieje duże prawdopodobieństwo, że instrukcje procedury zapisu danych do bufora wykonywane przez dwóch producentów będą również przeplatane. Oznacza to, że część danych umieszczona w trzecim elemencie bufora jest wyprodukowana przez proces Producent1, a pozostała przez proces Producent2. Analogiczna sytuacja może wystąpić w przypadku jednoczesnego pobierania danych z bufora przez dwóch lub więcej konsumentów.

Rozwiązanie tego problemu sprowadza się do zapewnienia — osobno dla producentów i konsumentów — wzajemnego wykluczania w dostępie do danego elementu bufora. W poniższym rozwiązaniu zastosowano dwa semafory binarne: jeden dla wzajemnego wykluczania producentów, drugi dla wzajemnego wykluczania konsumentów[5].

[5] Przedstawiona implementacja jest kompletna (gotowa do kompilacji), a liczba producentów i konsumentów jest określana podczas wykonywania programu (alokacja dynamiczna zadań).

```
with ADA.Text_IO; use ADA.Text_IO;
with ADA.Integer_Text_IO; use ADA.Integer_Text_IO;
with Semafory; use Semafory;

procedure ProducentKonsument is
    Max:Integer := 50;
    LProducentow: Integer := 5; LKonsumentow: Integer := 5;
    task type Producent(nr: Integer := 0);
    task type Konsument(nr: Integer := 0);
    type WskProducent is access Producent;
    type WskKonsument is access Konsument;
    producenci: array(1..LProducentow) of WskProducent;
    konsumenci: array(1..LKonsumentow) of WskKonsument;
    SP, SK: SemBin;  Miejsca, Dane: Sem;
    n: constant Integer := 10;
    Bufor: array(1..n) of Integer;
    wskP, wskK: Integer := 0;

  task body Producent is
    info: Integer := 1;
  begin
    loop
        Put("produkuje producent nr :"); Put(nr); New_Line;
      Miejsca.wait;
        SP.wait;
          wskP := (wskP mod n) + 1;
          Bufor(wskP) := info;
        SP.signal;
      Dane.signal;
    end loop;
  end Producent;
  task body Konsument is
    info: Integer := 1;
  begin
    loop
      Dane.wait;
        SK.wait;
          wskK := (wskK mod n) + 1;
          info := Bufor(wskK);
        SK.signal;
      Miejsca.signal;
        Put("konsumpcja konsumenta nr :"); Put(nr);
    end loop;
  end Konsument;
  procedure start is
    i, j: Integer;
  begin
    for i in 1..LProducentow loop
      producenci(i) := new Producent(i); end loop;
    for j in 1..LKonsumentow loop
      konsumenci(j) := new Konsument(j); end loop;
  end start;
begin
  Miejsca.init(n); -- wartość początkowa N
  Miejsca.init(0); -- wartość początkowa 0, bo bufor pusty
  Start;
end ProducentKonsument;
```

8.1.2. Spotkania

Ostatnie rozwiązanie przypomina problem symulacji semafora dwustronnie ograniczonego (zob. podrozdział 6.5) w oparciu o mechanizm spotkań. Innymi słowy, procedury umieszczania porcji danych oraz pobierania porcji danych mogłyby być przeniesione w przestrzeń adresową poniższego zadania, wykonującego instrukcję `select` z dwoma wejściami `Wstaw` i `Pobierz`. Dozory wejść zapewniają, że dane nie będą umieszczane w pełnym buforze i pobierane z pustego.

```
   n: constant Integer := 10;
task Bufor is
  entry Wstaw (X: in Typ);
  entry Pobierz(X: out Typ);
end Bufor;
task body Bufor is
  Buf: array (1..n) of Typ;
  wskP, wskK: Integer := 1;
  licznik: Integer := 0;
begin
  loop
    select
      when licznik < n =>
      accept Wstaw (X: in Typ) do
         Buf(wskP) := X;
         wskP := (wskP mod n) + 1;
         licznik := licznik + 1;
      end Wstaw;
  or
      when licznik > 0 =>
      accept Pobierz(X: out Typ) do
         X := Buf(wskK);
         wskK := (wskK mod n) + 1;
         licznik := licznik - 1;
      end Pobierz;
  end select;
end loop;
end Bufor;
```

Procesy producenta i konsumenta wstawiają i pobierają dane podczas spotkania z zadaniem `Bufor` odpowiednio w wejściu `Wstaw` i `Pobierz`.

```
task body Producent is
  info: Typ;
begin
  loop
    Produkuje_dane;
    Bufor.Wstaw(info);
  end loop;
end Producent;
task body Konsument is
  info: Typ;
begin
  loop
    Bufor.Pobierz(info);
```

```
    Konsumuje_dane;
  end loop;
end Konsument;
```

Powyższe rozwiązanie ma jedną podstawową wadę — w tym samym czasie zadanie `Bufor` może realizować spotkanie tylko z jednym z zadań, co w konsekwencji uniemożliwia jednoczesny dostęp producenta i konsumenta do różnych elementów bufora. Z tego powodu w tym rozwiązaniu uzyskano mniejszy stopień równoległego wykonania procesów w porównaniu z rozwiązaniem opartym na semaforach ogólnych i binarnych. Rozwiązanie oparte na semaforach zyskuje na znaczeniu wtedy, gdy dotyczy sytuacji, w której zadania przetwarzają duże porcje danych, tzn. czas względny wstawiania i pobierania porcji danych w stosunku do czasu realizacji operacji semaforowych jest znacznie większy.

8.1.3. Monitory

Sekwencyjne wykonanie operacji pobierania i umieszczania danych w buforze ma także miejsce wtedy, gdy wsparciem dla implementacji jest mechanizm monitora i obiektu chronionego. Zastosowanie mechanizmu monitora w implementacji *n*-elementowego bufora zilustrowano w poniższym kodzie[6]:

```
monitor Bufor is
  procedure Wstaw(X: in Typ);
  procedure Pobierz(X: out Typ);
  n: constant Integer := 10;
  bufor: array(0..n - 1) of Typ;
  ile: Integer := 0; WskP, WskK: Integer := 0;
  Miejsca, Dane: Warunek;
end Bufor;

monitor body Bufor is
  procedure Wstaw(X: in Typ) is
  begin
    if ile = n then wait(Miejsca); end if;
    WskP := (WskP + 1) mod n;
    bufor(WskP) := X;
    ile := ile + 1;
    if not empty(Dane) then signal(Dane); end if;
  end Wstaw;
  procedure Pobierz(X: out Typ) is
  begin
    if ile = 0 then wait(Dane); end if;
    WskK := (WskK + 1) mod n;
    X := bufor(WskK);
    ile := ile - 1;
    if not empty(Miejsca) then signal(Miejsca); end if;
  end Pobierz;
end Bufor;
```

[6] Operacje na zmiennych warunkowych `wait(...)`, `signal(...)`, `empty(...)` zostały opisane w punkcie 7.2.1 „Zmienne warunkowe”. Uproszczoną symulację działania mechanizmu monitora w Adzie wraz z implementacją powyższych operacji zawiera dodatek A.

Producenci i konsumenci, którzy chcą — odpowiednio — umieścić lub pobrać porcje danych z bufora, wywołują procedury monitora `Bufor.Wstaw(...)` i `Bufor.Pobierz(...)`. Zmienne warunkowe `Miejsca` i `Dane` pozwalają na wywłaszczenie z monitora producentów (operacja `wait(Miejsca)`), jeżeli bufor jest pełny, oraz konsumentów (operacja `wait(Porcje)`), jeżeli bufor jest pusty. Producenci są zawieszeni w kolejce zmiennej warunkowej `Miejsca`, konsumenci natomiast w kolejce zmiennej warunkowej `Dane`. Ostatnie instrukcje procedur `Wstaw` i `Pobierz` wznawiają potencjalnie zawieszone procesy w kolejkach zmiennych warunkowych, wykonując operacje — odpowiednio — `signal(Dane)`, która wznawia jednego z konsumentów, i `signal(Miejsca)`, która wznawia jednego z producentów. Funkcja `empty(Miejsca)` zapewnia, że operacja wznawiania producentów zawieszonych w kolejce zmiennej jest wykonywana tylko wówczas, gdy ta kolejka nie jest pusta. Analogiczne znaczenie ma funkcja `empty(Dane)` w stosunku do konsumentów.

8.1.4. Obiekty chronione

Rozwiązanie problemu producenta i konsumenta oparte na mechanizmie obiektu chronionego jest następujące:

```
type Elementy is array(Integer range <>) of Element;
n: constant Integer := 10;
protected type Bufor(n: Integer) is
  entry Pobierz(E: out Element);
  entry Wstaw(E: in Element);
private
  IndexWej, IndexWyj: Integer := 0;
  Ile: Integer := 0;   -- liczba porcji danych w buforze
  Buf: Elementy(1..n);
end Bufor;

protected body Bufor is
  entry Pobierz(E: out Element) when Ile > 0 is
  begin
    IndexWyj := (IndexWyj mod n) + 1;
    E := Buf(IndexWyj);
    Ile := Ile - 1;
  end Pobierz;
  entry Wstaw(E: in Element) when Ile < n is
  begin
    Buf(IndexWej) := E;
    IndexWej := (IndexWej mod n) + 1;
    Ile := Ile + 1;
  end Wstaw;
end Bufor;
procedure producenciKonsumenci is
    B1: Bufor(5);
  task type Producent;
  task body Producent is
    X:Element := 1;
  begin
    loop
        -- produkuje
      B1.Wstaw(X);
```

```
    end loop;
  end Producent;
  task type Konsument;
  task body Konsument is
    X: Element;
  begin
    loop
      B1.Pobierz(X);
      --  konsumuje
    end loop;
  end Konsument;
end producenciKonsumenci;
```

W powyższym rozwiązaniu bariery zdefiniowane dla wejść `Wstaw` i `Pobierz` gwarantują spełnienie założeń dotyczących współpracy producentów i konsumentów. Na tym etapie wiedzy Czytelnika, szczególnie po uwzględnieniu tematyki rozdziału 6. i 7., przykład ten jest na tyle prosty, że pominiemy jego szczegółowy opis.

Zaletą mechanizmu monitora i obiektu chronionego jest możliwość enkapsulacji i w rezultacie hermetyzacji zmiennych współdzielonych (bufor) i metod operujących na tych zmiennych. Oczywiście hermetyzacja przynosi te same korzyści co w przypadku programowania obiektowego. Zwiększa elastyczność (adaptowalność) zaimplementowanych programów, tzn. obiekt czy monitor może być bezpośrednio wykorzystany w wielu aplikacjach, ponieważ zmiana implementacji monitora (lub obiektu chronionego), przy zachowaniu nazw metod składowych (interfejsu), nie wpływa na modyfikacje kodu w aplikacjach, w których obiekt chroniony lub monitor został już wcześniej zastosowany.

Porównując implementację w oparciu o mechanizm monitora i obiektu chronionego, widać przewagę obiektu chronionego. Obliczanie barier dla wejść obiektu chronionego odbywa się przed ich wywołaniem i w niektórych zastosowaniach jest efektywniejszym rozwiązaniem w porównaniu ze zmiennymi warunkowymi monitora. W celu ilustracji rozważmy stan systemu, w którym bufor jest pełny i jeden z producentów wywołuje wejście lub procedurę — odpowiednio — obiektu chronionego i monitora[7]. W przypadku obiektu chronionego zostanie wykonana jedna operacja sprawdzenia warunku bariery i proces producenta zostanie zawieszony w kolejce do wejścia `Wstaw`. W przypadku mechanizmu monitora natomiast producent wejdzie do monitora, wykonując procedurę `Wstaw` (blokuje przy tym dostęp innym procesom do metod składowych monitora), następnie sprawdzi stan bufora, wykona operację `wait(Miejsca)`, co spowoduje jego wywłaszczenie z metody monitora, i zostanie zawieszony w kolejce warunku `Miejsca`.

8.1.5. Podsumowanie

Głównym celem niniejszego podrozdziału było pokazanie możliwości rozwiązań tego samego problemu w oparciu o różne mechanizmy synchronizacji, a dokładniej rzecz ujmując, chodziło o ilustrację efektywności implementacji tego samego algorytmu w oparciu

[7] Nie można tych wniosków uogólniać do każdego problemu programowania współbieżnego. Ilustrowały to przykłady rozwiązań problemu alokacji zasobów w podrozdziale 7.4. Również kolejne przykłady problemów współbieżnych pokażą, że to uogólnienie nie jest uzasadnione.

o różne mechanizmy synchronizacji zadań. Do oceny efektywności proponowanych rozwiązań (i w konsekwencji wyboru odpowiedniego mechanizmu synchronizacji) przyjęto trzy kryteria:

- stopień zrównoleglenia wykonania procesów (w przypadku środowiska wieloprocesorowego rzeczywistego zrównoleglenia);
- zdolność adaptacji przyjętego rozwiązania w innych programach, czyli elastyczność implementacji rozumianą jako łatwość dokonania zmian oraz ich weryfikacji w kontekście poprawności programów (np. liczba skutków ubocznych modyfikacji kodu);
- liczbę wykonywanych operacji związanych z protokołami synchronizacji procesów.

Każde z prezentowanych rozwiązań ma swoje wady i zalety. Zostały one opisane i wyjaśnione przy prezentacji każdego z nich. Poniższe podsumowanie pokazuje, że nie ma rozwiązania bez pewnych wad. Należy podkreślić, że wady i zalety danej implementacji nie wynikały z przyjętego algorytmu synchronizacji procesów (w każdym przypadku był stosowany ten sam algorytm), lecz z cech poszczególnych mechanizmów synchronizacji.

Podsumowując, na podstawie prezentowanych rozwiązań problemu producenta i konsumenta można sformułować dwa podstawowe wnioski:

1. Mechanizm semafora — największy stopień zrównoleglenia wykonywania procesów, ponieważ w tym samym czasie producent i konsument może pobierać porcje z bufora. W przypadku spotkań, monitora i obiektu chronionego operacje pobierania i wstawiania wykluczają się wzajemnie.
2. Obiekty chronione i monitory — łatwa adaptacja implementacji bufora w innych programach oraz elastyczność wynikająca z możliwości enkapsulacji metod i danych w jednej strukturze. Konsekwencje dotyczące wydajności metod testowania i weryfikacji programu dla strukturalnych mechanizmów są znane i występują nie tylko w programach współbieżnych (por. programowanie obiektowe).

Na zakończenie przedstawiono jeszcze jeden przykład rozwiązania problemu producenta i konsumenta w oparciu o mechanizm semafora. Celem tego rozwiązania jest zwiększenie stopnia równoległego wykonania procesów. Ponadto ten przykład uświadomi Czytelnikowi, że rozwiązania uzyskujące większy stopień zrównoleglenia wykonywania determinują znaczny wzrost możliwych przeplotów operacji atomowych, które z kolei komplikują proces weryfikacji poprawności programów współbieżnych.

Poprzednie rozwiązanie umożliwiało zadaniom producenta i konsumenta — odpowiednio — wstawianie i pobieranie danych z bufora w tym samym czasie. Założeniem poniższego rozwiązania jest umożliwienie jednoczesnego wstawiania porcji danych do różnych elementów bufora przez wielu producentów i analogicznie pobierania porcji danych z różnych elementów przez konsumentów.

```
-- definicje bufora, semaforów ogólnych i binarnych
-- oraz ich początkowa inicjalizacja jak w poprzednich przykładach
task body Producent is
  wsklok: Integer := 0;
  info: Typ;
begin
  loop
      Produkuje;
    Miejsca.wait;
      SP.wait;
        wskP := (wskP mod n) + 1;
        wsklok := wskP;
      SP.signal;
            -- semafor nie wyklucza jednoczesnego wstawiania danych do różnych elementów bufora
      Bufor(wsklok) := info;
    Dane.signal;
  end loop;
end Producent;

task body Konsument is
  wsklok: Integer := 0;
  info: Typ;
begin
  loop
    Dane.wait;
      SK.wait;
        wskK := (wskK mod n) + 1;
        wsklok := wskK;
      SP.signal;
           -- semafor nie obejmuje operacji pobierania porcji
      Info := Bufor(wskK);
    Miejsca.signal;
      Konsumuje;
  end loop;
end Konsument;
```

Dużą rolę w tym rozwiązaniu pełnią zmienne lokalne. Gwarantują one zapamiętanie przez każdego z producentów indeksu wolnego miejsca w buforze oraz przez konsumenta indeksu miejsca, w którym są zapisane dane. Rozważmy stan programu, w którym dwóch producentów `Producent1` i `Producent2` próbuje jednocześnie umieszczać porcje danych w buforze. Załóżmy, że drugi i trzeci element bufora jest wolny, stąd w rozważanym stanie `wskP = 1`, tzn. wskazuje na pierwszy element bufora.

Przeplot operacji wykonywanych przez zadania `Producent1` i `Producent2` może być następujący:

- zadanie `Producent1` przypisuje `wskP = 2` oraz zapamiętuje wartość `wskP` w zmiennej lokalnej `wsklok = wskP` (semafor gwarantuje niepodzielność tych dwóch operacji);
- zadanie `Producent1` rozpoczyna operację zapisywania danych do bufora — `Bufor(2);`
- zadanie `Producent2` przypisuje `wskP = 3` oraz zapamiętuje wartość zmiennej `wskP` w zmiennej lokalnej `wsklok = wskP`;

- zadanie `Producent2` rozpoczyna operację zapisywania danych do bufora — `Bufor(2)`.

Łatwo zauważyć, że implementacja oparta jedynie na jednej zmiennej globalnej `wskP` naruszałaby własność wzajemnego wykluczania w dostępie do poszczególnych elementów bufora. Analogiczny przeplot operacji ma miejsce w przypadku jednoczesnego pobierania porcji danych przez dwóch konsumentów. Zmienne lokalne pozwalają zapamiętać indeks elementu bufora, do którego dane zadanie wstawia lub z którego pobiera dane. Oczywiście zysk wynikający z równoległości operacji wstawiania i pobierania danych z bufora jest tym większy, im większa jest różnica pomiędzy czasem zapisu danych a czasem wykonania operacji przypisania `wsklok = wskP`.

Jednak poniższe rozwiązanie jest prawidłowe w szczególnych przypadkach. Przeanalizujmy poprawność wykonania powyższego programu w heterogenicznym, wieloprocesorowym systemie złożonym z procesorów o różnych prędkościach. W takim systemie prędkość wykonywania operacji przez procesy jest zależna od tego, który procesor został przydzielony do ich wykonania.

Rozważmy stan, w którym bufor jest pusty i jeden z konsumentów jest zawieszony na operacji opuszczania semafora `Dane`. Dwóch producentów `Producent1` i `Producent2` realizuje powyżej opisany przeplot operacji i jednocześnie umieszcza porcje — odpowiednio — w pierwszym i drugim elemencie bufora. Zauważmy, że w przypadku gdy proces `Producent2` szybciej zapisuje dane do bufora (w wyniku przydziału szybszego procesora) niż proces `Producent1`, to `Producent2` może podnieść semafor `Dane` umożliwiający dostęp konsumenta do bufora. Jednak w wyniku takiego przeplotu operacji konsument będzie pobierał dane z elementu bufora, w którym są jeszcze zapisywane dane przez proces `Producent1`.

Dodatkowym zabezpieczeniem w powyższym rozwiązaniu mogą być dynamicznie zmieniające się priorytety zadań w zależności od operacji wykonywanej przez określone zadanie — np. to, które wcześniej zaczęło wykonywać operacje protokołu wstępnego (czyli wcześniej przypisze zmiennej lokalnej numer wolnego miejsca), ma wyższy priorytet. Jednak to projektujący aplikacje musi określić, czy zastosowanie dodatkowego algorytmu nadawania priorytetu poszczególnym procesom jest uzasadnione w porównaniu z zyskiem związanym ze zwiększeniem stopnia zrównoleglenia operacji.

8.2. Problem pięciu filozofów

Problem ucztujących filozofów należy do klasycznych problemów programowania współbieżnego. Został on omówiony w podrozdziale 3.3 „Żywotność globalna" w kontekście współzawodnictwa procesów, które może prowadzić do stanu braku żywotności globalnej programu. Jednak do tej pory nie przedstawiono poprawnej implementacji tego problemu, co stanowi główny cel tego podrozdziału. O ile w poprzednim podrozdziale skupiono się na uzyskaniu efektywnej implementacji tego samego algorytmu synchronizacji procesów, stosując różne mechanizmy synchronizacji, o tyle celem niniejszego jest ocena mechanizmu synchronizacji w kontekście przyjętego algorytmu rozwiązania tego samego problemu współbieżnego.

Każdy z filozofów wykonuje naprzemiennie dwie operacje — myślenie i jedzenie. Struktura procesu filozofa jest następująca:

```
task Filozof;
task body Filozof is
begin
  loop
      Filozof_myśli;
    Protokół_wstępny;
       Filozof_je;
    Protokół_końcowy;
  end loop;
end Filozof;
```

Przed każdym z pięciu filozofów stoi talerz oraz leżą dwa widelce. Filozof potrzebuje dwóch widelców (dwóch zasobów), aby móc rozpocząć operację „jedzenie", która stanowi sekcję krytyczną. Problem polega na zsynchronizowaniu dostępu filozofów do widelców (każdy widelec jest współdzielony przez dwóch filozofów siedzących obok siebie) gwarantującego żywotność lokalną i globalną programu.

Współdzielenie widelców przez sąsiadujących filozofów wymusza wzajemne wykluczanie w ich użytkowaniu. Naturalnym mechanizmem gwarantującym wzajemne wykluczanie jest semafor binarny, dlatego w pierwszym rozwiązaniu problemu dostęp do każdego widelca będzie synchronizowany przez semafor binarny. Każdy filozof, aby rozpocząć jedzenie, musi wykonać operację opuszczania dwóch semaforów binarnych. Przy wyjściu z sekcji krytycznej filozof oddaje widelce, realizując operację `signal` — kolejno dla widelca z prawej, a następnie z lewej strony.

```
Widelce: array(1..5) of SemBin;
task type Filozof(Nr: Integer);
task body Filozof is
begin
  loop
      Filozof_myśli;
    Widelce(Nr).wait;
    Widelce((Nr mod 5) + 1).wait;
       Filozof_je;
    Widelce(nr).signal;
    Widelce((Nr mod 5) + 1).signal;
  end loop;
end Filozof;
```

Łatwo zauważyć, że rozwiązanie umożliwiające filozofom kompletowanie widelców przez ich sekwencyjne pobieranie może doprowadzić do blokady w przypadku, gdy każdy z filozofów podniesie lewy widelec i blokując dostęp do niego, oczekuje na prawy widelec. Problem blokady procesów filozofów został szczegółowo przedstawiony w podrozdziale 3.3. Omówiono metody unikania blokady i zapobiegania blokadzie oparte na negacji jednego z czterech warunków koniecznych wystąpienia blokady: wzajemnego wykluczania, wywłaszczania procesów z zasobów dzielonych, częściowego przydziału zasobów oraz cyklicznego oczekiwania procesów.

Z wymagań założonych na współpracę procesów w problemie pięciu filozofów wynika, że nie można zwiększyć liczby dostępnych zasobów, dlatego negacja warunku wzajemnego wykluczania nie będzie podstawą poniższych implementacji. Negacja warunku

wywłaszczania procesów wymaga dodatkowego procesu, który identyfikowałby wystąpienie blokady i czasowo wywłaszczał jeden z procesów (filozofów) z zasobu dzielonego (oddanie widelca przez filozofa). To podejście bazuje na nieefektywnych metodach wykrywania i likwidacji blokad i także zostanie pominięte.

Prezentowane w tym podrozdziale rozwiązania zapobiegają wystąpieniu stanu blokady w wyniku negacji jednego z dwóch warunków: warunku częściowego przydziału zasobów — w czasie oczekiwania na zajęty zasób proces nie zwalnia zasobu przydzielonego w poprzedniej operacji, oraz warunku czekania cyklicznego — istnieje zamknięty łańcuch procesów, z których każdy oczekuje na zasoby przetrzymywane przez poprzednika. W problemie pięciu filozofów potencjalne spełnienie warunku częściowego przydziału wynika z następującego założenia: filozof przetrzymuje widelec w oczekiwaniu na kolejny widelec. Potencjalne spełnienie warunku cyklicznego oczekiwania wynika natomiast ze struktury, jaką tworzą procesy filozofa i widelce, ponieważ pierwszy widelec jest wykorzystywany przez pierwszego i piątego filozofa.

Negacja jednego z dwóch warunków wystąpienia blokady jest osiągana dzięki konstrukcji co najmniej dwóch różnych algorytmów synchronizacji procesów. Poniżej przedstawiono implementację tych algorytmów w oparciu o mechanizm semafora, monitora, obiektu chronionego oraz mechanizm spotkań.

8.2.1. Semafory

W pierwszym rozwiązaniu wyróżniono dwa rodzaje procesów filozofa. Filozofowie z numerami nieparzystymi będą kolejno podnosić prawy, a następnie lewy widelec, a filozofowie o numerach parzystych podnoszą widelce w odwrotnej kolejności. Takie postępowanie filozofów gwarantuje, że nie będzie spełniony warunek cyklicznego czekania, ponieważ nie może powstać zamknięty łańcuch żądań zasobowych (żądań dostępu do widelców) procesów (punkt 3.3.4).

```
Widelce: array(1..5) of SemBin;
task type FilozofN(Nr: Integer);
task type FilozofP(Nr: Integer);
task body FilozofP is        -- dotyczy filozofów o nr 2 i 4
begin
  loop
      Filozof_myśli;
    Widelce((Nr mod 5) + 1).wait;
    Widelce(Nr).wait;
       Filozof_je;
    Widelce(nr).signal;
    Widelce((Nr mod 5) + 1).signal;
  end loop;
end FilozofP;

task body FilozofN is        -- dotyczy filozofów o nr 1, 3 i 5
begin
  loop
      Filozof_myśli;
```

```
    Widelce(Nr).wait;
    Widelce((Nr mod 5) + 1).wait;
       Filozof_je;
    Widelce((Nr mod 5) + 1).signal;
    Widelce(nr).signal;
  end loop;
end FilozofN;
```

W tym rozwiązaniu są spełnione własności żywotności lokalnej i globalnej. Rozważmy najbardziej niebezpieczny stan, w którym każdy z procesów w tym samym czasie chce podnieść swój pierwszy widelec: `FilozofN1` podnosi prawy widelec o numerze 1, `FilozofP2` podnosi lewy widelec o numerze 3, `FilozofN3` nie może podnieść prawego widelca o numerze 3; `FilozofP4` podnosi swój lewy widelec o numerze 5, `FilozofN5` nie może podnieść zajętego lewego widelca. Widelce o numerach 2 i 4 są wolne i `FilozofN1` oraz `FilozofP4` mogą podnieść widelce i rozpocząć jedzenie. Łatwo pokazać, że jakikolwiek przeplot operacji podnoszenia widelców w powyższym programie pozostawia dwa wolne widelce, co gwarantuje zarówno brak blokady, jak i zagłodzenia procesów.

Najczęściej prezentowane w literaturze symetryczne (ze względu na jednakową strukturę procesów filozofa) rozwiązanie, gdzie każdy proces filozofa podnosi prawy, a następnie lewy widelec, polega na ograniczeniu do czterech liczby filozofów jednocześnie współzawodniczących o dostęp do widelców. To ograniczenie uzyskano poprzez definicję semafora ogólnego `Jadalnia` o wartości początkowej równej 4. Nawet w szczególnym przypadku jednoczesnego współzawodnictwa czterech filozofów jeden z nich będzie miał możliwość podniesienia dwóch widelców i wykonania sekcji krytycznej „jedzenie".

```
procedure Filozofow5 is
    task type Filozof(Nr: Integer := 0);
    type wskF is access Filozof;
    Filozofowie: array(1..5) of wskF;
    Widelce: array(1..5) of SemBin;
    Jadalnia: Sem;  -- wartość początkowa semafora 4
  task body Filozof is
  begin
    loop
        Filozof_myśli;
      Jadalnia.wait;
      Widelce(Nr).wait;
      Widelce((Nr mod 5) + 1).wait;
        Filozof_je;
      Widelce(nr).signal;
      Widelce((Nr mod 5) + 1).signal;
      Jadalnia.signal;
    end loop;
  end Filozof;
begin
  Jadalnia.init(4);
  for i in 1..5 loop Filozofowie(i) := new Filozof(i);
  end loop;
end Filozofow5;
```

8.2.2. Monitory

Kolejne rozwiązanie jest oparte na mechanizmie monitora. Własność mechanizmu monitora pozwala w naturalny sposób zanegować warunek częściowego przydziału. Filozofowie podnoszą widelce tylko w przypadku, gdy oba są wolne.

```
monitor Jadalnia;
    Wolne: array(0..4) of Integer range 0..2 := (others => 2);
    jedzenie: array(0..4) of Warunek;
  procedure BioreWidelce(I: Integer) is
  begin
    if Wolne(I) < 2 then wait(jedzenie(i));
    end if;
    Wolne((I + 4) mod 5) := Wolne((I + 4) mod 5) - 1;
    Wolne((I + 1) mod 5) := Wolne((I + 1) mod 5) - 1;
  end BioreWidelce;
  procedure OddajeWidelce(I: Integer) is
  begin
    Wolne((I + 4) mod 5) := Wolne((I + 4) mod 5) + 1;
    Wolne((I + 1) mod 5) := Wolne((I + 1) mod 5) + 1;
    if Wolne((I + 1) mod 5) = 2 then
       if not empty(Signal(jedzenie((I + 1) mod 5)))
          then Signal(jedzenie((I + 1) mod 5)); end if;
    end if;
    if Wolne((I + 4) mod 5) = 2 then
      if not empty(Signal(jedzenie((I + 4) mod 5)))
        Signal(jedzenie((I + 4) mod 5)); end if;
    end if;
  end OddajeWidelce;
end Jadalnia;

task type Filozof(Nr: Integer);
task body Filozof is
begin
  loop
      Filozof_myśli;
    BioreWidelece(Nr);
      Filozof_myśli;
    OddajeWidelce(Nr);
  end loop;
end Filozof;
```

I-ty element tablicy `Wolne` określa liczbę dostępnych widelców dla *i*-tego filozofa. Dodatkowo zdefiniowano tablicę `jedzenie` typu `Warunek`, która stanowi deklarację pięciu zmiennych warunkowych. Zmienne warunkowe umożliwiają zawieszenie procesów filozofa w oczekiwaniu na wolne widelce. Wykonanie powyższego programu jest następujące: *i*-ty filozof wywołuje procedurę monitora `BioreWidelce` i sprawdza w niej stan *i*-tego elementu tablicy `Wolne`. Jeżeli nie są dostępne dwa widelce (`Wolne(i) < 2`), to wykonuje operację `wait(jedzenie(i))` i zostaje zawieszony w kolejce warunku `jedzenie(i)`, a jednocześnie odblokowuje dostęp do monitora dla innych procesów. Sąsiadujący filozof, który odłoży drugi z brakujących widelców, wykonuje operację `signal(jedzenie(i))` i wznawia wykonanie procesu zawieszonego w kolejce do zmiennej warunkowej `jedzenie(i)`. Krytyczna dla poprawności i efektywności powyższego rozwiązania jest

własność mechanizmu monitora, która gwarantuje, że procesy zawieszone w kolejkach zmiennych warunkowych są wznawiane w pierwszej kolejności w stosunku do procesów oczekujących w kolejce wejściowej monitora (punkt 7.2.1).

Efektywne rozwiązanie oparte na semaforach — negujące warunek częściowego przydziału — jest trudne do realizacji, ponieważ zadanie nie może wycofać się z operacji opuszczania semafora. Może jedynie wielokrotnie testować wartości elementów tablicy `Wolne`.

```
Wolne: array(0..4) of Integer range 0..2 := (others => 2);
s: SemBin;
procedure BioreWidelce(I: Integer) is
begin
  loop
     s.wait; -- wielokrotnie ta operacja może mieć efekt pusty
       if Wolne(I) < 2 then
         s.signal;
       else begin
         Wolne((I + 4) mod 5) := Wolne((I + 4) mod 5) - 1;
         Wolne((I + 1) mod 5) := Wolne((I + 1) mod 5) - 1;
         s.signal;
            end;
       end if;
  end loop;
end BioreWidelce;
```

Filozof wywołuje procedurę `BioreWidelce`, w sekcji krytycznej tej procedury sprawdza stan widelców i w przypadku, gdy co najmniej jeden z widelców jest zajęty, opuszcza sekcję krytyczną (podnosi semafor `s`), co umożliwia innym zadaniom sprawdzenie dostępności widelców. Mała efektywność tego rozwiązania jest związana z aktywnym oczekiwaniem zadań na dostęp do widelców, co sprowadza się do wielokrotnego wykonywania operacji opuszczania i podnoszenia semafora binarnego `s` w przypadku, gdy widelce są zajęte. Łatwo zauważyć, że to rozwiązanie dopuszcza możliwość zagłodzenia jednego z filozofów, nawet wtedy, gdy semafor `s` ma zaimplementowaną kolejkę typu FIFO dla zadań oczekujących na operację opuszczania semafora.

Lepszym rozwiązaniem jest implementacja podobna do symulacji działania monitora za pomocą semaforów. Jednak w tym przypadku liczba semaforów oraz liczba dodatkowych zmiennych (określających to, ilu filozofów oczekuje na możliwość podniesienia dwóch widelców) jest zależna od liczby filozofów. Dla klasycznego przypadku pięciu filozofów potrzebujemy: pięciu semaforów dla każdego widelca, pięciu zmiennych określających liczbę zawieszonych zadań na każdym z semaforów i dodatkowo semafor binarny gwarantujący wzajemne wykluczanie w dostępie do tablicy `Wolne`. W tym podejściu procedura `BioreWidelce` może być zapisana w następujący sposób:

```
Wolne: array(0..4) of Integer range 0..2 := (others => 2);
jedzenie: array(0..4) of SemBin;
   -- dla zdefiniowanego pakietu w podrozdziale 6.5 inicjalizacja powinna być
   -- w programie głównym := (others => 0)
licznik: array(0..4) of Integer := (others => 0);
s: SemBin;
   -- semafor binarny dla ochrony dostępu do tablicy Wolne
procedure BioreWidelce(I: Integer) is
```

```
begin
  loop
    s.wait;
    if Wolne(I) < 2 then czekaj(I); end if;
    Wolne((I + 4) mod 5) := Wolne((I + 4) mod 5) - 1;
    Wolne((I + 1) mod 5) := Wolne((I + 1) mod 5) - 1;
    s.signal;
  end loop;
end BioreWidelce;
    ........
   -- analogiczna procedura OddajeWidelce
procedure Czekaj(I: Integer) is
begin
    licznik(I) := licznik(I) + 1;
    s.signal;
    jedzenie(I).wait;
    licznik(I) := licznik(I) - 1;
end Czekaj;
procedure Sygnalizuj(I: Integer) is
begin
  if licznik(I) > 0 then
    jedzenie(I).signal;
  else
    s.signal;
  end if;
end Sygnalizuj;
```

Semafor s zapewnia wzajemne wykluczanie w dostępie do tablicy `Wolne`. Na początku procedury `BioreWidelce` *i*-ty filozof, opuszczając semafor `s`, blokuje możliwość sprawdzenia przez pozostałe zadania dostępności widelców (analogicznie jak w monitorze). Jeżeli widelce nie są dostępne, wywołuje procedurę `Czekaj(I)`, w której jest zawieszany na operacji opuszczania semafora `jedzenie(I)`. Z kolei *j*-ty filozof, odkładając swoje widelce, sprawdza dostępność widelców swoich sąsiadów i ewentualnie podnosi semafory `jedzenie((J + 1) mod 5)` i/lub `jedzenie((J - 1) mod 5)` — w przypadku gdy są dostępne oba widelce oraz gdy są na tych semaforach zawieszone zadania filozofów. Liczbę zawieszonych zadań na semaforze `jedzenie(I)` określa zmienna `licznik(I)`. Łatwo zauważyć, że operacja `Czekaj(I)` i `Sygnalizuj(I)` są podobne do operacji wznawiania `signal` i wstrzymywania `wait` zadań zawieszonych w kolejkach zmiennych warunkowych.

To dość skomplikowane rozwiązanie jest efektywne, ponieważ nie ma aktywnego oczekiwania (poprzez wielokrotne podnoszenie i opuszczanie semafora) na spełnienie warunku dostępu do dwóch widelców. W porównaniu do rozwiązania opartego na mechanizmie monitora, w którym była zdefiniowana tylko jedna lokalna tablica warunków (wewnątrz monitora), w powyższym rozwiązaniu musimy zdefiniować dwie globalne tablice liczników dla każdego oczekującego filozofa oraz tablicę semaforów.

8.2.3. Obiekty chronione

Rozwiązanie problemu pięciu filozofów oparte na mechanizmie obiektu chronionego jest zdecydowanie trudniejsze niż w przypadku monitora, w szczególności gdy rozwiązanie ma zapewniać negację warunku częściowego przydziału zasobów oraz hermetyzację

w jednym obiekcie chronionym zasobów dzielonych (widelców) oraz zmiennych synchronizujących zadania filozofów. Wynika to stąd, że w warunku dozoru danego wejścia można jedynie korzystać ze zmiennych globalnych lub zdefiniowanych wewnątrz obiektu chronionego. Oznacza to, że nie jest możliwy następujący zapis.

```
type Tablica is array(Integer range <>) of Integer;
protected type Jadalnia is
  entry BioreWidelce(I: in Integer);
  entry OddajeWidelce(I: in Integer);
private
Wolne: Tablica(0..4) := (others => 2);
end Jadalnia;

protected body Jadalnia is
  entry BioreWidelce(I: in Integer)
            when Wolne(I) = 2 is -- błąd - niedostępne w Adzie
                 -- parametr aktualny wywoływania wejścia nie może być testowany
                 -- w warunku logicznym bariery
  begin
    if Wolne(I) < 2 then return; end if;

    Wolne((I + 4) mod 5) := Wolne((I + 4) mod 5) - 1;
    Wolne((I + 1) mod 5) := Wolne((I + 1) mod 5) - 1;
  end BioreWidelce;
  .........
end Jadalnia;
```

Parametr aktualny wywołania wejścia nie może być zastosowany do obliczania warunku dozoru, ponieważ są one obliczane przed wywołaniem wejścia obiektu chronionego. Gdybyśmy zastosowali zmienną globalną `I`, to musielibyśmy zagwarantować wyłączny dostęp do tej zmiennej (np. za pomocą semafora binarnego). Ale i w tym przypadku powstaje problem, kiedy zwolnić dostęp do zmiennej dzielonej. Najbardziej narzucającym się rozwiązaniem jest umieszczenie instrukcji `S.signal` (przy założeniu, że semafor `S` synchronizuje dostęp do zmiennej `I`). Jednak wywołanie wejścia któregokolwiek zadania (operacja `S.signal` dla zadania typu semaforowego w `entry BioreWidelce...`) w treści metody obiektu chronionego jest operacją potencjalnie prowadzącą do blokady (operacje tego typu zostały opisane w punkcie 7.3.4)[8].

```
protected body Jadalnia is
  entry BioreWidelce (I: in Integer) when Wolne(I) = 2 is
  begin
    ........
    S.Signal;  -- operacja potencjalnie prowadząca do blokady
  end BioreWidelce;
    ........
```

Rozwiązaniem jest zdefiniowanie dodatkowego obiektu chronionego `S`, który zagwarantuje wzajemne wykluczanie w dostępie do zmiennej `I`. Realizacja procedury `Sygnalizuj` tego obiektu pozwala na dostęp do zmiennej `I` i może być wywołana w wejściu `entry BioreWidelce(I: in Integer)...`.

[8] Jeżeli zostanie zidentyfikowany stan niebezpieczny (czyli potencjalnie prowadzący do blokady), zostanie zgłoszony wyjątek `Program_Error`.

```
........
I: Integer := 0;
protected S is
  entry Czekaj;
  procedure Sygnalizuj;
private
  Dostepna: Boolean := True;
end S;
protected body S is
  entry Czekaj when Dostepna is
  begin
    Dostepna := False;
  end Czekaj;
  procedure Sygnalizuj is
  begin
    Dostepna := True;
  end Sygnalizuj;
end S;
protected Jadalnia is
  entry BioreWidelce (j: in Integer);
  ........
end Jadalnia;
protected body Jadalnia is
  entry BioreWidelce(j: in Integer) when Wolne(I) = 2 is
                                -- zmienna globalna I
  begin
    Wolne((j + 4) mod 5) := Wolne((j + 4) mod 5) - 1;
    Wolne((j + 1) mod 5) := Wolne((j + 1) mod 5) - 1;
    S.Sygnalizuj;   --  zwalnia dostęp do zmiennej I
 end BioreWidelce;
..........

task type Filozof(Nr: Integer);
task body Filozof is
begin
  loop
      Filozof_myśli;
    S.Czekaj; -- blokuje dostęp do zmiennej I
    I := Nr;
    Jadalnia.BioreWidelce(I);
      Filozof_je;
    Jadalnia.OddajeWidelce(I);
  end loop;
end Filozof;
procedure glowna is

type Tablica is array(Integer range <>) of Integer;

I: Integer := 0;

protected S is
  entry Czekaj;
  procedure Sygnalizuj;
private
  Dostepna: Boolean := True;
end S;
```

```
protected body S is
  entry Czekaj when Dostepna is
  begin
    Dostepna := False;
  end Czekaj;
  procedure Sygnalizuj is
  begin
    Dostepna := True;
  end Sygnalizuj;
end S;

protected Jadalnia is
  entry BioreWidelce (j: in Integer);
 -- ........
 private
      Wolne: Tablica(0..4) := (others => 2);
end Jadalnia;

protected body Jadalnia is
  entry BioreWidelce(j: in Integer) when Wolne(I) = 2 is
                                -- zmienna globalna I
  begin
    Wolne((j + 4) mod 5) := Wolne((j + 4) mod 5) - 1;
    Wolne((j + 1) mod 5) := Wolne((j + 1) mod 5) - 1;
    S.Sygnalizuj;   -- zwalnia dostęp do zmiennej I
 end BioreWidelce;
--.........
end Jadalnia;

task type Filozof(Nr: Integer);

task body Filozof is
begin
  loop
      --Filozof_myśli;
    S.Czekaj; -- blokuje dostęp do zmiennej I
    I := Nr;
    Jadalnia.BioreWidelce(I);
      --Filozof_je;
    --Jadalnia.OddajeWidelce(I);
  end loop;
end Filozof;
begin
      null;
end glowna;
```

Powyższe rozwiązanie gwarantuje bezpieczeństwo programu, jednak nie można go uznać za poprawne, ponieważ zawieszone zadanie na wejściu `BioreWidelce` będzie blokowało dostęp do zmiennej `I` dla innych zadań. Oznacza to, że może wystąpić przeplot, w którym jeden filozof je, drugi oczekuje na wejście do jadalni (łatwo zauważyć, że jest to sąsiad filozofa obecnie jedzącego), a pozostali nie mogą konkurować o dostęp do obiektu chronionego `Jadalnia` (pomimo że dwa z trzech widelców w jadalni są wolne), ponieważ są zawieszeni na operacji `S.Czekaj`. W tym wypadku nie jest spełnione następujące wymaganie: jeżeli są wolne dwa odpowiednie widelce, to przy braku współzawodnictwa filozof powinien mieć możliwość natychmiastowego ich pobrania.

Najprostszym rozwiązaniem jest specyfikacja w obiekcie chronionym `Jadalnia` procedury `BioreWidelce` i `OddajeWidelce` dla każdego filozofa.

```
type Tablica is array(Integer range <>) of Integer;
protected type Jadalnia is
  entry BioreWidelce1;
  entry BioreWidelce2;
  entry BioreWidelce3;
  entry BioreWidelce4;
  entry BioreWidelce5;
  procedure OddajeWidelce1;
    ........
  entry OddajeWidelce5;
private
  Wolne: Tablica(0..4) := (others => 2);
end Jadalnia;

protected body Jadalnia is
  entry BioreWidelce1 when Wolne(0) = 2 is
  begin
    Wolne(1) := Wolne(1) - 1;
    Wolne(4) := Wolne(4) - 1;
  end BioreWidelce1;
  entry BioreWidelce2 when Wolne(1) = 2 is
  begin
    Wolne(0) := Wolne(0) - 1;
    Wolne(2) := Wolne(2) - 1;
  end BioreWidelce2;
    ........
  procedure OddajeWidelce1 is
  begin
    Wolne(4) := Wolne(4) + 1;
    Wolne(1) := Wolne(1) + 1;
  end OddajeWidelce1;
    ........
  procedure OddajeWidelce5 is
  begin
    Wolne(0) := Wolne(0) + 1;
    Wolne(3) := Wolne(3) + 1;
  end OddajeWidelce5;
end Jadalnia;
```

Jak widać, powyższe rozwiązanie jest mało elastyczne i nie jest reprezentatywne dla modelu programowania serwerów, ponieważ zmiana liczby filozofów (klientów) wymaga znacznych zmian w specyfikacji i treści obiektu chronionego. Poprawnym i naturalnym rozwiązaniem jest zastąpienie powyższej deklaracji 10 wejść deklaracją dwóch rodzin wejść. Poniżej zaprezentowano ogólną strukturę tego rozwiązania, szczegóły implementacji pozostawiono Czytelnikowi.

```
subtype Numer is Integer Range 1..5;
protected Jadalnia is
  entry BioreWidelce(Numer);
  entry OddajeWidelce(Numer);
private
  ........
end Jadalnia;
```

```
protected body Jadalnia is
  entry BioreWidelce(for I in Numer)
        when warunek(I) is
  begin
    ........
  end BioreWidelce;
  entry OddajeWidelce(for I in Numer)
        when warunek(I) is
  begin
    ........
  end OddajeWidelce;
end Jadalnia;
```

Inne nasuwające się rozwiązanie w oparciu o obiekt chroniony sprowadza się do deklaracji tablicy obiektów, z których każdy reprezentuje widelec, co z kolei uniemożliwia jednoczesne podniesienie dwóch różnych widelców przez filozofów. Jednak w tym przypadku bez dodatkowej synchronizacji jest możliwe wystąpienie blokady. W poprawnym rozwiązaniu należy zastosować dodatkową współdzieloną zmienną, na przykład semafor ogólny lub obiekt chroniony, która pozwoli na jednoczesne współzawodnictwo o widelce co najwyżej czterem filozofom. Należy zauważyć, że to rozwiązanie jest w rzeczywistości tym samym co rozwiązanie oparte na mechanizmie semafora, tyle tylko że z zastosowaniem obiektu chronionego. Innymi słowy, w tym rozwiązaniu żadna własność obiektu chronionego nie zwiększa jakości tego rozwiązania w porównaniu z pierwszym prezentowanym w tym punkcie, dlatego też pominięto jego implementację.

Wsparciem dla rozwiązań opartych na obiektach chronionych, w szczególności tych negujących warunek częściowego przydziału, jest instrukcja rekolejkowania — `requeue`. Wewnątrz obiektu chronionego, w zależności od stanu programu (w omawianym przykładzie od stanu wykorzystania zasobów —widelców), instrukcja ta pozwala ewentualnie przenieść zadanie do kolejki innego wejścia obiektu chronionego. Ten typ synchronizacji został szczegółowo omówiony w podrozdziale 7.4 przy opisie rozwiązań problemu alokacji zasobów. Adaptację rozwiązań problemu alokacji zasobów — opartych na instrukcji rekolejkowania — dla rozwiązania problemu pięciu filozofów pozostawiono Czytelnikowi.

8.2.4. Spotkania

Ostatnie prezentowane rozwiązanie jest oparte na bezpośredniej implementacji obsługi dostępu do widelców oraz zabezpieczenia przed wystąpieniem stanu blokady dzięki instrukcji selektywnego oczekiwania omówionej w podrozdziale 6.2[9].

W poniższym rozwiązaniu założono, że dostęp do widelców kontroluje zadanie `Kontrola_Widelcow` z deklaracją dwóch wejść `Podnies` i `Odloz`.

```
procedure Filozofow5 is
  package Czynnosci is
    procedure Mysli;
```

[9] Przedstawiona implementacja jest kompletna (gotowa do kompilacji), ponieważ procesy filozofa są tworzone podczas wykonywania programu (alokacja dynamiczna zadań).

```
  procedure Je;
end Czynnosci;
package body Czynnosci is
  procedure Mysli is
  begin
    delay 2.0;
  end Mysli;
  procedure Je is
  begin
    delay 3.0;
  end Je;
 end Czynnosci;

N: constant := 5;
type Liczba_Filozofow is range 0..N - 1;
task type Fil(P: Liczba_Filozofow);
task type Kontrola_Widelcow is
  entry Podnies;
  entry Odloz;
end Kontrola_Widelcow;
```

Zapewnienie żywotności globalnej (brak blokady) jest realizowane poprzez negację warunku cyklicznego czekania — dopuszczonych jest jedynie czterech filozofów do jednoczesnego współzawodnictwa o dostęp do widelców.

```
task Brak_Blokady is
  entry Wchodzi;
  entry Opuszcza;
end Brak_Blokady;
type Filozof is access Fil;
Widelce: array(Liczba_Filozofow) of Kontrola_Widelcow;
Filozofowie: array(Liczba_Filozofow) of Filozof;
-- pierwsze rozwiązanie
task body Kontrola_Widelcow is
begin
  loop
    accept Podnies;
    accept Odloz;
  end loop;
end Kontrola_Widelcow;
task body Brak_Blokady is
  Max: constant Integer := N - 1;
  L_Jedzacy: Integer range 0..Max := 0; -- liczba jedzących filozofów
begin
  loop
    select
      when L_Jedzacy < Max =>
      accept Wchodzi do
        L_Jedzacy := L_Jedzacy + 1;
      end Wchodzi;
     or
      accept Opuszcza do
        L_Jedzacy := L_Jedzacy - 1;
      end Opuszcza;
    end select;
  end loop;
end Brak_Blokady;
```

Każde z zadań filozofa jest tworzone dynamicznie z odpowiednią wartością wyróżnika z zakresu od 0 do 4.

```
    task body Fil is
      Widelec1, Widelec2: Liczba_Filozofow;
      use Czynnosci;
    begin
      Widelec1 := P;
      Widelec2 := (Widelec1 mod N) + 1;
      loop
           Mysli;
        Brak_Blokady.Wchodzi;
        Widelce(Widelec1).Podnies;
        Widelce(Widelec2).Podnies;
           Je;
        Widelce(Widelec1).Odloz;
        Widelce(Widelec2).Odloz;
        Brak_Blokady.Opuszcza;
      end loop;
    end Fil;
  begin
    for P in Liczba_Filozofow loop
      Filozofowie(P) := new Fil(P);
    end loop;
  end Filozofow5;
  with Ada.Text_io; use Ada.Text_IO;
  procedure Filozofow5 is

    package Czynnosci is
      procedure Mysli;
      procedure Je;
    end Czynnosci;
    package body Czynnosci is
      procedure Mysli is
      begin
        delay 2.0;
          put_Line("Mysli");
      end Mysli;
      procedure Je is
      begin
        delay 3.0;
                put_Line("Je");
      end Je;
     end Czynnosci;

  N: constant := 5;
    type Liczba_Filozofow is range 0..N - 1;
    task type Fil(P: Liczba_Filozofow);
    task type Kontrola_Widelcow is
      entry Podnies;
      entry Odloz;
    end Kontrola_Widelcow;

        task Brak_Blokady is
      entry Wchodzi;
      entry Opuszcza;
    end Brak_Blokady;
```

```
  type Filozof is access Fil;
  Widelce: array(Liczba_Filozofow) of Kontrola_Widelcow;
  Filozofowie: array(Liczba_Filozofow) of Filozof;
  -- pierwsze rozwiązanie
  task body Kontrola_Widelcow is
  begin
    loop
      accept Podnies;
      accept Odloz;
    end loop;
  end Kontrola_Widelcow;
  task body Brak_Blokady is
    Max: constant Integer := N - 1;
    L_Jedzacy: Integer range 0..Max := 0; -- liczba jedzących filozofów
  begin
    loop
      select
        when L_Jedzacy < Max =>
        accept Wchodzi do
          L_Jedzacy := L_Jedzacy + 1;
        end Wchodzi;
       or
        accept Opuszcza do
          L_Jedzacy := L_Jedzacy - 1;
        end Opuszcza;
      end select;
    end loop;
  end Brak_Blokady;
task body Fil is
    Widelec1, Widelec2: Liczba_Filozofow;
    use Czynnosci;
  begin
    Widelec1 := P;
    Widelec2 := (Widelec1 + 1) mod N;
    loop
         Mysli;
      Brak_Blokady.Wchodzi;
      Widelce(Widelec1).Podnies;
      Widelce(Widelec2).Podnies;
         Je;
      Widelce(Widelec1).Odloz;
      Widelce(Widelec2).Odloz;
      Brak_Blokady.Opuszcza;
    end loop;
  end Fil;
begin
  for P in Liczba_Filozofow loop
    Filozofowie(P) := new Fil(P);
  end loop;
end Filozofow5;
```

W tego typu rozwiązaniach opartych na specyfikacji zadań typu serwer awaria jednego z serwerów jest niedopuszczalna. Wykonanie zadań klienta nie powinno mieć natomiast wpływu na poprawność działania programu, lecz co najwyżej na jego efektywność. Awaria jednego z zadań filozofa w sekcji lokalnej (`Mysli`) nie ma wpływu na wykonanie pozostałych, jednak awaria jednego z zadań w sekcji krytycznej powoduje zabloko-

wanie dostępu do widelców dla pozostałych zadań filozofów. Jeżeli można określić maksymalny czas wykonywania sekcji krytycznej przez zadanie filozofa, to poniższa specyfikacja zadania `Kontrola_Widelcow` gwarantuje ciągłość wykonywania zadań w przypadku awarii jednego z nich.

```
-- drugie rozwiązanie
  task body Kontrola_Widelcow is
  begin
    loop
      select
        accept Podnies;
      or
        terminate;
      end select;
      select
        accept Odloz;
      or
        delay 4.0; -- maksymalny czas wykonywania sekcji krytycznej
      end select;
    end loop;
  end Kontrola_Widelcow;
```

8.2.5. Podsumowanie

Cechy mechanizmów synchronizacji opisane podczas omawiania problemu producenta i konsumenta, które zapewniały większą elastyczność rozwiązań, są aktualne dla problemu pięciu filozofów. W przykładach rozwiązań problemu producenta i konsumenta, stosując jeden algorytm synchronizacji procesów, ocenie poddano efektywność implementacji w zależności od rodzaju stosowanego mechanizmu synchronizacji procesów. Podstawowym kryterium oceny zaproponowanych rozwiązań był stopień zrównoleglenia operacji wykonywanych przez procesy wstawiające dane do bufora i pobierające dane z bufora.

W problemie pięciu filozofów szczególną uwagę skupiono na zagwarantowaniu żywotności globalnej, co wynikało bezpośrednio ze specyfiki wymagań dotyczących synchronizacji filozofów. Podstawą rozwiązań zapewniających brak blokady była negacja jednego z dwóch koniecznych warunków dla wystąpienia blokady w programie: negacji warunku cyklicznego oczekiwania oraz negacji warunku częściowego przydziału zasobów. Z prezentowanych przykładów wynika, że wybór algorytmu determinował wybór mechanizmu synchronizacji. W przypadku negacji warunku cyklicznego czekania łatwość i efektywność implementacji gwarantował mechanizm semafora ogólnego oraz mechanizm spotkań. Wymienione mechanizmy natomiast — zastosowane w algorytmie negacji warunku częściowego przydziału zasobów — oraz w szczególności mechanizm obiektu chronionego generowały bardzo skomplikowane i mało czytelne kody. Z kolei rozwiązanie oparte na negacji warunku częściowego przydziału wspierał mechanizm klasycznego monitora dzięki możliwości zawieszania i wznawiania procedur monitora.

Podsumowując, celem tego podrozdziału było pokazanie, że pewne mechanizmy synchronizacji są dedykowane dla implementacji przyjętego algorytmu synchronizacji procesów. Wybór mechanizmu jest kluczowy dla osiągnięcia poprawnego i jak najprostszego zapisu danego algorytmu synchronizacji procesów.

8.3. Problem pisarzy i czytelników

Trzecim z klasycznych problemów programowania współbieżnego jest problem pisarzy i czytelników. Czytelnia w tym problemie reprezentuje zasób dzielony, jednak dostęp do niej nie jest określony klasyczną regułą wzajemnego wykluczania 1 z *n*. W problemie pisarzy i czytelników występują dwie klasy procesów: czytelnicy, którzy cyklicznie odczytują dane z czytelni, oraz pisarze, którzy zapisują dane do czytelni. Wielu czytelników może jednocześnie odczytywać dane, zapisywanie danych przez pisarza wyklucza natomiast możliwość przebywania w tym samym czasie w czytelni zarówno czytelnika, jak i innego pisarza. Struktury zadań reprezentujących pisarza i czytelnika są następujące:

```
task body Czytelnik is
  begin
    loop
        Sekcja_lokalna;
      Protokół_wstępny;
        Czytanie;
      Protokół_końcowy;
    end loop;
end Czytelnik;
task body Pisarz is
  begin
    loop
        Sekcja_lokalna;
      Protokół_wstępny;
        Pisanie;
      Protokół_końcowy;
end loop;
end Pisarz;
```

Rozwiązanie tego problemu sprowadza się do zsynchronizowania grup procesów (pisarzy i czytelników) współzawodniczących o dostęp do czytelni, gwarantującego brak blokady i brak zagłodzenia jednej z grup procesów. O ile w problemie pięciu filozofów na pierwszy plan wysuwał się problem blokady, o tyle w problemie pisarzy i czytelników należy szczególną uwagę zwrócić na stan zagłodzenia pisarzy przez czytelników, którego prawdopodobieństwo wystąpienia wzrasta wraz ze wzrostem liczby czytelników.

Problem pisarzy i czytelników jest abstrakcją reguł dostępu w bazach danych, w których zakłada się możliwość jednoczesnego czytania danych przez wiele procesów, jednak z drugiej strony wymaga wzajemnego wykluczania podczas modyfikacji danych w celu zapewnienia ich spójności. W większości rzeczywistych systemów (w szczególności w systemach zarządzania bazami danych) procesy czytające zawartość obiektów dzielonych żądają i uzyskują dostęp do tych obiektów wielokrotnie częściej niż procesy modyfikujące ich stan. Przykładem może być prosty system zarządzania bazą biblioteki, gdzie w tym samym czasie wielu czytelników przegląda wiele tytułów, zanim dokona konkretnej rezerwacji. Rezerwacja określonego tytułu musi podlegać wzajemnemu wykluczaniu, ponieważ jest zmieniany status książki z dostępnej na zarezerwowaną. Przykładów systemów tego typu jest wiele, dlatego często rozwiązanie problemu synchronizacji w rzeczywistych systemach skupia się na możliwości zagłodzenia procesów modyfikujących stan współdzielonych zasobów, czyli w omawianym abstrakcyjnym problemie — procesów pisarzy przez czytelników.

8.3.1. Semafory

W pierwszym rozwiązaniu synchronizacja procesów pisarzy i czytelników jest oparta na dwóch semaforach: jednym ogólnym, którego wartość określa liczbę wolnych miejsc w czytelni, oraz drugim binarnym, zapewniającym wzajemne wykluczanie pisarzy w dostępie do czytelni.

```
LC: Integer := 10; LP: Integer := 3;     -- liczba czytelników i pisarzy
pisarze: array(1..LP) of Pisarz;
czytelnicy: array(1..LC) of Czytelnik;
Miejsca, Wolne: Sem := LC;               -- liczba miejsc w czytelni
P: SemBin;                               -- wzajemne wykluczanie pisarzy
task body Czytelnik is
begin
  loop
      Put("Sekcja lokalna");
    Miejsca.wait;
      Put("Czytanie");
    Miejsca.signal;
  end loop;
end Czytelnik;
task body Pisarz is
  i: Integer;
begin
  loop
      Put("Sekcja lokalna");
    P.wait;
    for i in 1..LC loop Wolne.wait; end loop;
      Put("Pisanie");
    for i in 1..LC loop Wolne.signal; end loop;
    P.signal;
  end loop;
end Pisarz;
```

W rozwiązaniu założono, że liczba czytelników jest równa liczbie miejsc w czytelni. Opuszczenie semafora `Miejsca` umożliwia czytelnikowi wejście do czytelni. Pisarz stopniowo rezerwuje wszystkie miejsca w czytelni, ograniczając w niej liczbę czytelników. Po zajęciu wszystkich miejsc pisarz wchodzi do czytelni. Semafor `S` gwarantuje wzajemne wykluczanie pisarzy już na etapie rezerwacji miejsc w czytelni, tzn. w tym samym czasie tylko jeden pisarz rezerwuje miejsca w czytelni. Jednak to rozwiązanie ma kilka wad:

- Brak gwarancji, że po wyjściu pisarza z czytelni wszyscy czytelnicy wejdą do czytelni, zanim kolejny pisarz zapisze nowe dane. Wynika to stąd, że po wyjściu pisarza z czytelni wszystkie miejsca są wolne, lecz mogą być one zarówno zajmowane przez wchodzących czytelników, jak i blokowane przez innych pisarzy.
- Liczba operacji niezbędnych do synchronizacji zadań jest zależna od liczby czytelników. Liczba czytelników determinuje liczbę operacji opuszczania semafora `Miejsca` przez pisarza, ponieważ musi on zarezerwować wszystkie miejsca, zanim wejdzie do czytelni.

Rozwiązanie pozbawione powyższych wad zaproponował P.B. Hansen[10]. Każdy proces, który chce wejść do czytelni, sprawdza, czy są procesy z drugiej grupy oczekujące na wejście. Jeśli ich nie ma, umożliwia wejście do czytelni wszystkim procesom ze swojej grupy. Wstrzymanie procesów czytelnika, jeśli pisarz jest w czytelni, jest realizowane przez semafor `Czytanie`. Semafor `Pisanie` wstrzymuje natomiast pisarzy, gdy w czytelni przebywają czytelnicy. Po wyjściu z czytelni ostatniego procesu z danej grupy wpuszczane są procesy z drugiej. Pisarz po opuszczeniu czytelni podnosi wielokrotnie semafor `Czytelnia` (operacja `Czytelnia.signal`). Analogicznie postępuje opuszczający czytelnię ostatni czytelnik, realizując operacje na semaforze `Pisanie`. Pisarze wchodzą do czytelni, wzajemnie się wykluczając (tę własność gwarantuje dodatkowy semafor binarny). Kolejny semafor binarny gwarantuje wzajemne wykluczanie w dostępie do zmiennych globalnych (dzielonych) określających liczbę oczekujących pisarzy i czytelników. Kod ilustrujący to rozwiązanie został zamieszczony w dodatku A.

Dość duży stopień skomplikowania powyżej przedstawionych rozwiązań wynika z braku bezpośredniej implementacji w klasycznym semaforze metody umożliwiającej sprawdzenie liczby zadań oczekujących na operację opuszczania semafora[11]. W Adzie istnieje możliwość sprawdzania liczby zadań zawieszonych (wywołujących wejścia zadania typu serwer) w oczekiwaniu na realizację spotkania z zadaniem serwera.

8.3.2. Spotkania

Struktura rozwiązania problemu pisarzy i czytelników oparta na mechanizmie spotkań została omówiona w punkcie 6.2.2. Zaprezentowane rozwiązanie nie gwarantowało, że po zapisie danych przez pisarza wszyscy czytelnicy zdążą je odczytać, zanim kolejny pisarz wejdzie do czytelni. Kompilacja tego rozwiązania z prezentowanym w punkcie 6.2.2 pozwala uzyskać kompletny kod opisywanego przypadku. Poniższy kod oprócz ogólnej struktury zadania `Czytelnia` (np. pominięto programowanie wejść dla tego zadania) zawiera jedynie instrukcje zapewniające odczyt wszystkim czytelnikom oraz specyfikacje i treść zadań `Pisarz` i `Czytelnik`.

```
task Czytelnia is
  entry StartCzytanie(E: out typ);
  entry StopCzytanie;
  entry Pisanie(E: in typ);
end Czytelnia;
task body Czytelnia is
iluCzyta: Integer := 0;    -- liczba czytelników w czytelni
ksiazka: typ;
begin
  loop
    select
      when Pisanie'Count = 0 =>
      accept StartCzytanie(E: out typ) do...
    or
      accept StopCzytanie do...
    or
```

[10] Hansen P., *Podstawy systemów operacyjnych*, WNT, Warszawa 1979.

[11] Na wyznaczenie i testowanie aktualnej wartości semafora pozwalają semafory zdefiniowane w systemie Unix.

```
      when iluCzyta = 0 =>
      accept Pisanie(E: in typ) do...
    ........
      loop
        select
          accept StartCzytanie(E: out typ) do
            iluCzyta := iluCzyta + 1;
            E := ksiazka;
          end StartCzytanie;        -- wpuszczenie wszystkich  oczekujących czytelników
        else exit;                  -- natychmiastowe czytanie lub wyjście z pętli
        end select;
      end loop;
    end select;
  end loop;
end Czytelnia;
task Czytelnik;
task body Czytelnik is
begin
  loop
    Czytelnia.StartCzytanie(...);
       -- czyta
    Czytelnia.StopCzytanie;
       -- sekcja lokalna
  end loop;
end Czytelnik;
task Pisarz;
task body Pisarz is
begin
  loop
       -- sekcja lokalna
    Czytelnia.Pisanie(...);
  end loop;
end Pisarz;
```

8.3.3. Monitory

Synchronizację zadań pisarzy i czytelników można efektywnie zaimplementować w oparciu o mechanizmy wyższego poziomu niż semafory, takie jak monitory i obiekty chronione. Wynika to stąd, że tak jak w przypadku spotkań, w tych mechanizmach istnieją metody umożliwiające zawieszenie danego zbioru procesów w oczekiwaniu na spełnienie pewnego warunku. W przypadku monitorów są to zmienne warunkowe oraz metody `wait` i `signal` operujące na tych zmiennych.

```
monitor Czytelnia is
  Lczyt, Lpisz: Integer := 0;          -- liczba czytelników i pisarzy w czytelni
  Czytanie, Pisanie: warunek;
  procedure WchodziCzytelik is
  begin
    if Lpisz > 0 or not empty(Pisanie) then
      wait(Czytanie);
    end if;
    Lczyt := Lczyt + 1;
    signal(Czytanie);
  end WchodziCzytelik;
  procedure WychodziCzytelnik is
```

```
    begin
      Lczyt := Lczyt - 1;
      if Lczyt = 0 then
        signal(Pisanie);
      end if;
    end WychodziCzytelnik;
    procedure Pisze is
    begin
      if Lczyt > 0 or Lpisz > 0 then
        wait(Pisanie);
      end if;
      Lpisz := 1;
          -- pisze
      Lpisz := 0;
      if not empty(Czytanie) then signal(Czytanie);
      else signal(Pisanie);
      end if;
    end Pisze;
  end Czytelnia;
  task Czytelnik;
  task body Czytelnik is
  begin
    loop
      Czytelnia.WchodziCzytelnik;
          -- czyta
      Czytelnia.WychodziCzytelnik;
          -- sekcja lokalna
  end loop;
  end Czytelnik;
  task Pisarz;
  task body Pisarz is
  begin
    loop
          -- sekcja lokalna
      Czytelnia.Pisze;
    end loop;
  end Pisarz;
```

W powyższym rozwiązaniu dwie zmienne warunkowe `pisanie` i `czytanie` określają stan czytelni — to, czy jest ona zajęta przez czytelników, czy przez pisarza. Jeżeli czytelnicy nie mogą wejść do czytelni, to są zawieszani w kolejce warunku `czytanie` (wykonują instrukcję `wait(czytanie)`) i reaktywowani przez wychodzącego z czytelni pisarza instrukcją `signal(czytanie)`. Analogicznie, jeżeli czytelnicy przebywają w czytelni, to pisarz jest zawieszany w kolejce warunku `pisanie` i wznawiany (`signal(Pisanie)`) przez ostatniego czytelnika wychodzącego z czytelni.

8.3.4. Obiekty chronione

Jeszcze bardziej naturalne i przede wszystkim efektywniejsze rozwiązanie (ze względu na liczbę operacji synchronizujących dostęp do czytelni) zapewnia obiekt chroniony.

```
package Czytelnicy_Pisarze is
  procedure Czytaj(I: out Typ);
   procedure Zapisz(I: Typ);
```

```
end Czytelnicy_Pisarze;
package body Czytelnicy_Pisarze is
  procedure Czytaj_Plik(I: out Typ) is
  begin
    ........
  end Czytaj_Plik;
  procedure Zapisz_Plik(I: Typ) is
  begin
    ........
  end Zapisz_Plik;

  protected Kontrola is
    entry Zacznij_Czytac;
    procedure Zakoncz_Czytac;
    entry Zacznij_Zapisywac;
    procedure Zakoncz_Zapisywac;
  private
    Czytelnicy: Natural := 0;
    Pisarze: Boolean := False;
  end Kontrola;

  procedure Czytaj(I: out Typ) is
  begin
    Kontrola.Zacznij_Czytac;
    Czytaj_Plik(I);
    Kontrola.Zakoncz_Czytac;
  end Czytaj;
  procedure Zapisz(I: Typ) is
  begin
    Kontrola.Zacznij_Zapisywac;
    Zapisz_Plik(I);
    Kontrola.Zakoncz_Zapisywac;
  end Zapisz;

  protected body Kontrola is
    entry Zacznij_Czytac when not Pisarze and
                          Zacznij_Zapisywac'Count = 0 is
    begin
      Czytelnicy := Czytelnicy + 1;
    end Zacznij_Czytac;
    procedure Zakoncz_Czytac is
    begin
      Czytelnicy := Czytelnicy - 1;
    end Zakoncz_Czytac;
    entry Zacznij_Zapisywac when not Pisarze and Czytelnicy = 0 is
    begin
      Pisarze := True;
    end Zacznij_Zapisywac;
    procedure Zakoncz_Zapisywac is
    begin
      Pisarze := False;
    end Zakoncz_Zapisywac;
  end Kontrola;
end Czytelnicy_Pisarze;
```

Bariery dla wejść wykluczają jednoczesne przebywanie w czytelni pisarzy i czytelników. Ponadto warunek `Zacznij_Zapisywac'Count = 0` dla wejścia `Zacznij_Czytac` gwarantuje brak zagłodzenia pisarzy, ponieważ czytelnik jest zablokowany na wejściu do czytelni w przypadku, gdy którykolwiek pisarz oczekuje na wywołanie wejścia `Zacznij_Zapisywac`.

8.3.5. Podsumowanie

Z powyższych prezentowanych rozwiązań wynika, że problem pisarzy i czytelników jest abstrakcją problemów rzeczywistych, w których istnieje prawdopodobieństwo zagłodzenia jednej z grup procesów, w szczególności dotyczy to zagłodzenia pisarzy. Dane mechanizmy — m.in. spotkań, monitor, obiektu chronionego — pozwalają określić liczbę procesów czekających na wejście do czytelni. Ta własność mechanizmów w efektywny i w prosty sposób pozwalała zaimplementować reguły gwarantujące żywotność lokalną programu. Brak tej własności w przypadku semafora generował natomiast skomplikowany kod rozwiązania.

Na podstawie rozwiązań trzech klasycznych problemów współbieżnych można sformułować ogólny wniosek, że dany mechanizm synchronizacji jest dedykowany dla konkretnych klas problemów programowania współbieżnego. Innymi słowy, dany mechanizm synchronizacji efektywnie i w naturalny sposób rozwiązuje pewną klasę problemów, jest natomiast pozbawiony tych cech dla innej klasy problemów współbieżnych. Dobrym przykładem ilustrującym tę własność jest mechanizm obiektu chronionego zastosowany w implementacji bufora w problemie producenta i konsumenta, w implementacji czytelni dla problemu pisarzy i czytelników oraz w implementacji problemu pięciu filozofów. Dla pierwszych dwóch problemów implementacja była najefektywniejsza (w sensie liczby niezbędnych operacji synchronizujących procesy) oraz najbardziej elastyczna w sensie łatwości adaptacji w innych aplikacjach w porównaniu z innymi mechanizmami synchronizacji. W przypadku problemu pięciu filozofów natomiast obiekt chroniony okazał się mechanizmem najmniej efektywnym.

Ponadto przyjęte rozwiązanie dla danego problemu współbieżnego ma wpływ na wybór odpowiedniego mechanizmu synchronizacji. W problemie pięciu filozofów rozważaliśmy dwa możliwe rozwiązania negujące jeden z koniecznych warunków wystąpienia blokady: warunek cyklicznego oczekiwania lub warunek częściowego przydziału. Zastosowanie semafora ogólnego pozwalało na efektywną implementację gwarantującą negację warunku cyklicznego oczekiwania. Negacja warunku częściowego przydziału zasobów w oparciu o mechanizm semafora była natomiast skomplikowana i mało elastyczna. Z kolei monitor dzięki możliwości zawieszania procesów w kolejkach związanych ze zmiennymi warunkowymi w prosty i przejrzysty sposób pozwalał na implementację negacji warunku częściowego przydziału zasobów.

8.4. Ćwiczenia i zadania

1. Zaproponuj implementację automatu komórkowego znanego jako „Gra w życie” w oparciu o następujące mechanizmy synchronizacji: semafory oraz obiekty chronione.

Zbiór komórek jest ułożony w prostokątną tablicę tak, że każda komórka ma ośmiu sąsiadów (poziomo, pionowo i po przekątnych). Każda komórka jest żywa lub martwa. Mając początkowy, skończony zbiór żywych komórek, oblicz konfigurację uzyskaną po ciągu pokoleń. Reguły przechodzenia od jednego pokolenia do następnego są następujące:

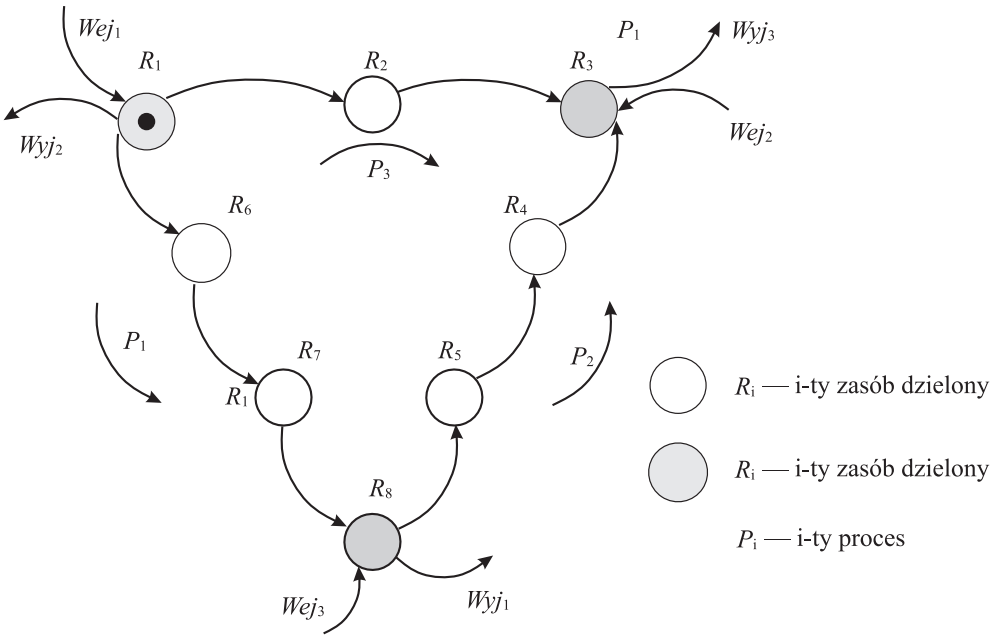
- jeżeli komórka jest żywa i ma mniej niż dwóch żywych sąsiadów, to umiera;
- jeżeli ma dwóch lub trzech żywych sąsiadów, to żyje nadal;
- jeżeli ma czterech lub więcej żywych sąsiadów, to umiera;
- martwa komórka z dokładnie trzema żywymi sąsiadami staje się żywa.

Każda komórka jest symulowana przez proces. Należy rozwiązać dwa problemy. Po pierwsze, wyliczanie następnego pokolenia musi być zsynchronizowane, tzn. modyfikacja każdej komórki musi uwzględniać stan sąsiednich komórek w tym samym pokoleniu. Po drugie, należy stworzyć dużą strukturę danych z procesami i kanałami komunikacyjnymi.

2. Stosując jedynie semafory binarne, zsynchronizuj procesy producenta i konsumenta w dostępie do ograniczonego bufora. Udowodnij poprawność programu. Czy istnieje możliwość poprawnego rozwiązania problemu wielu producentów i wielu konsumentów umieszczających dane w buforze i pobierających je z niego tylko z użyciem semaforów binarnych? Jeśli tak, zaimplementuj rozwiązanie.

3. Linia produkcyjna (przetwarzanie potokowe). W systemie są trzy marszruty wytwarzania elementów (rysunek 8.2). Jedna marszruta produkcyjna stanowi ciąg zasobów dzielonych i może jednocześnie pomieścić tyle obrabianych elementów, ile jest zasobów. W systemie są procesy nakładające nieobrobione elementy i procesy zdejmujące już przetworzone elementy, odpowiednio na wejściu i wyjściu danej marszruty. Ponadto z każdym zasobem jest związany jeden proces. Pierwszy z procesów realizuje pierwszą fazę obróbki, kolejny drugą fazę itd. W czasie oczekiwania elementu na zajęty zasób element nie zwalnia zasobu przydzielonego do wykonywania poprzedniej fazy obróbki. Napisz program ilustrujący powyższy schemat przetwarzania elementów. Zwróć uwagę, że trzy zasoby są współdzielone przez elementy z różnych marszrut.

4. Lotniskowiec ma pokład mogący jednocześnie pomieścić `N` samolotów. Jedyny pas startowy umożliwia samolotom startowanie i lądowanie, jednak w tym samym czasie może z niego korzystać tylko jeden samolot. Gdy liczba samolotów na lotniskowcu jest mniejsza niż `K` ($0 < K < N$), priorytet w dostępie do pasa startowego mają samoloty lądujące, w przeciwnym razie startujące. Zapisz algorytm samolotu, który funkcjonuje według schematu postój – start – lot – lądowanie. Skonstruuj algorytm synchronizacji procesów samolot oraz zapisz go w oparciu o mechanizm semafora, spotkań oraz obiektu chronionego. Porównaj i oceń te implementacje.



Rysunek 8.2. *Model systemu produkcyjnego*

Skorowidz

I

K

M

N

O

P

R

S

T

U

W

Z

Ż

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Przyszłość informatyki to przetwarzanie współbieżne. Stać Cię na pozostanie w tyle?

Coraz niższe ceny i powszechna dostępność sprzętu komputerowego o architekturze wieloprocesorowej powodują, że umiejętność projektowania i budowania aplikacji przetwarzających informacje współbieżnie staje się wręcz niezbędna każdemu zawodowemu programiście. W większości współczesnych języków programowania bezpośrednio zaimplementowano metody tworzenia zadań wykonywanych równolegle oraz wysokopoziomowe mechanizmy komunikacji i synchronizacji procesów.

Tworzenie efektywnych aplikacji współbieżnych wciąż jednak wymaga dużej, specjalistycznej wiedzy dotyczącej systemów operacyjnych oraz programowania nisko- i wysokopoziomowego, o czym przekonało się wielu studentów kierunków informatycznych i profesjonalnych programistów. Na szczęście teraz wszyscy mogą sięgnąć po książkę *Programowanie współbieżne. Systemy czasu rzeczywistego*". Pomoże ona uniknąć wielu typowych błędów związanych z tworzeniem aplikacji współbieżnych i pokaże, jak rozwiązywać problemy specyficzne dla tej dziedziny. Lektura ułatwi też zdobycie praktycznej umiejętności projektowania architektury niezawodnego współbieżnego oprogramowania, a także przybliży wiedzę na temat mechanizmów i metod wykorzystywanych przy tworzeniu systemów równoległych czasu rzeczywistego.

- Podstawowe pojęcia dotyczące programowania współbieżnego
- Opis metod weryfikacji poprawności programów współbieżnych
- Definicje i własności mechanizmów synchronizacji oraz komunikacji
- Przykłady rozwiązań problemów programowania współbieżnego
- Opis mechanizmów wspierających programowanie systemów czasu rzeczywistego
- Implementacja metod szeregowania zadań w systemach czasu rzeczywistego
- Opis metod i mechanizmów języka Ada 2005 umożliwiających implementację programów współbieżnych i systemów czasu rzeczywistego

Paweł Majdzik — od 1998 roku pracuje jako adiunkt w Instytucie Sterowania i Systemów Informatycznych Uniwersytetu Zielonogórskiego. Jest autorem bądź współautorem ponad trzydziestu opracowań naukowych: książek, artykułów, referatów wydanych w kraju i za granicą, dotyczących informatyki, w szczególności związanych z analitycznymi metodami modelowania i projektowania systemów współbieżnych.

helion.pl
księgarnia internetowa

Nr katalogowy: 7721

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-4302-8

Cena: 49,00 zł

9 788324 643028