

**skry
pty**

WYDANO W ROKU 40-LECIA
POLITECHNIKI WROCŁAWSKIEJ

ZBIGNIEW HUZAR

WSTĘP
DO PROGRAMOWANIA
WSPÓŁBIEŻNEGO

WROCŁAW 1985

POLITECHNIKA
WROCŁAWSKA

skrypty

W SPRZEDAŻY:



- JELEŃIEWSKI T., PAWĘSKA R., *Laboratorium teorii automatów*, Wrocław 1969
- JELEŃIEWSKI T., SIELICKI A., *Projektowanie urządzeń cyfrowych*, Wrocław 1977
- KABACIK T., *Ćwiczenia laboratoryjne z urządzeń telekomunikacyjnych, cz. I, Urządzenia teletransmisyjne*, Wrocław 1979
- KARKOWSKI Z., WOŹNIAK M., *Zasady konstrukcji elektronicznej aparatury pomiarowej*, Wrocław 1976
- KAZIMIERCZAK J., *Elementy syntezy formalnej systemów operacyjnych*, Wrocław 1979
- KAZIMIERCZAK W., *Sieci teletechniczne*, Wrocław 1975
- KOMOROWSKI W., PAWĘSKA R., *Zbiór zadań z teorii automatów. Teoria komputerów*, Wrocław 1979
- KULISZEWSKI T., *Teletransmisyjne systemy impulsowe*, Wrocław 1975
- Laboratorium automatyki. Praca zbiorowa pod red. B. CHOROWSKIEGO*, wyd. II popr., Wrocław 1975
- Laboratorium elementów elektronicznych przyrządów pomiarowych. Praca zbiorowa pod red. H. KARKOWSKIEJ*, Wrocław 1976
- MILKOWSKA M., *Telefonia automatyczna*, Wrocław 1972
- Projektowanie urządzeń cyfrowych wspomagane komputerem. Ćwiczenia laboratoryjne, cz. I. Symulacja sieci logicznych. Praca zbiorowa pod red. A. SIELICKIEGO*, Wrocław 1977
- REŃOWSKI J., *Laboratorium z akustyki psychofizjologicznej*, Wrocław 1972
- REŃOWSKI J., *Akustyka psychofizjologiczna. Ćwiczenia laboratoryjne*, wyd. II popr., Wrocław 1974
- SIELICKI A., JELEŃIEWSKI T., *Podstawy metodologii projektowania*, Wrocław 1978
- SZEŁOCH R. F., *Podstawy niezawodności elementów elektronicznych*, Wrocław 1976
- TALARCZYK E., *Podstawy techniki ultradźwięków*, Wrocław 1979
- Zbiór zadań i problemów z teorii sterowania. Praca zbiorowa pod red. Z. BUBNICKIEGO*, Wrocław 1979

40,—
46,—
11,—
13,—
12,—
35,—
7,—
24,—
12,—
20,—
14,—
20,—
13,—
18,—
28,—
26,—
28,—
16,—
16,—
17,—
28,—
14,—
51,—
16,—

Politechnika Wrocławska



Zbigniew Huzar

**Wstęp
do programowania
współbieżnego**

Wrocław 1985

Skrypt do przedmiotu
Programowanie współbieżne
na Wydziale Elektroniki
i na Wydziale Informatyki i Zarządzania

Opiniodawca
Andrzej BABORSKI

Opracowanie redakcyjne i korekta
Ewa BIELECKA-JONKO



2429324/3

© Copyright by Wydawnictwo Politechniki Wrocławskiej, Wrocław 1985

WYDAWNICTWO POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

Nakład 375 +50 egz. Ark. wyd. 9,5. Ark. druk. 97/8. Papier offset. kl. V, 70 g, B1.
Oddano do druku w styczniu 1985 r. Druk ukończono w lutym 1985 r.
Zakład Graficzny Politechniki Wrocławskiej. Zam. nr 109/85 – Z-7 – Cena zł 95,–

Proc. 102/K/85

PRZEDMOWA

Programowanie jako dziedzina nauki liczy około 30 lat, w ciągu których weszło na drogę szybkiego, wręcz gwałtownego rozwoju. Rozwój ten jest powodowany z jednej strony - szybkim rozwojem technologii przemysłu komputerowego, z drugiej - wzrostem potrzeb, czy też oczekiwań użytkowników. Jednym z ważnych przejawów rozwoju programowania są narodziny pojęcia programowania równoległego (współbieżnego), stanowiącego uogólnienie klasycznego programowania sekwencyjnego. Pojawienie się nowych języków programowania równoległego dało użytkownikom komputerów nowe uniwersalne narzędzie umożliwiające rozwiązywanie szerszej klasy problemów.

Mówiąc ogólnie o programowaniu należałoby przedstawić narzędzie - język programowania oraz sposób posługiwania się tym narzędziem - metodologię programowania. W skrypcie podano wprowadzenie do zagadnień programowania równoległego i główny nacisk położono na narzędzie - dokonano przeglądu najważniejszych kierunków w tym zakresie - natomiast zamiast metodologii użytkowania przedstawiono ilustracyjne przykłady zastosowań. Wydaje się, że taki wybór zagadnień jest uzasadniony, gdyż stosowana praktycznie metodologia programowania (rozumiana jako zespół metod specyfikacji problemów, metod konstrukcji i analizy programów) ciągle pozostaje bardziej dziedziną sztuki niż nauki. Należy jednak zaznaczyć, że problemy te stają się przedmiotem intensywnych badań.

Sposób prezentacji materiału jest nieformalny, dotyczy to zarówno syntaktyki jak i semantyki języków. Zostało to podyktowane chęcią przedstawienia w ograniczonej objętości, bez formalnego balastu, możliwie szerokiego spektrum zagadnień języków programowania. Z tego też względu skrypt nie jest poświęcony żadnemu z przedstawionych języków.

Skrypt jest przeznaczony dla studentów lat wyższych informatyki. Może być także fragmentami przydatny dla osób, które dopiero zamierzają sięgnąć po nowy rodzaj narzędzia programistycznego i pragną uzyskać informacje wstępne. Wystarczającym warunkiem swobodnego czytania skryptu jest znajomość programowania sekwencyjnego oraz - przynajmniej ogólna - znajomość języka Pascal. Szczególnie użyteczny będzie skrypt dla osób, które zetknęły się z problematyką projektowania systemów operacyjnych.

Rozdziały 1. i 2. skryptu stanowią część wprowadzającą, pozostałe natomiast są ich rozwinięciem i mogą być studiowane niezależnie; zaleca się jednak, aby czytanie rozdziałów 4. (opis języka Modula), 6. (opis języka Ada) i 7. (opis języka CHILL) było poprzedzone zapoznaniem się z rozdziałem 3. (opis języka Concurrent Pascal).

P.S. W okresie ponad trzech lat, który upłynął od chwili złożenia maszynopisu skryptu w Wydawnictwie a przekazaniem go do druku ukazała się w 1982 r., nakładem WNT, książka W. Iszkowskiego i M. Manieckiego "Programowanie współbieżne", której zakres tematyczny częściowo pokrywa się z zakresem niniejszego skryptu. Polecając gorąco uwagę Czytelników tę książkę, należy zwrócić uwagę na różnice pomiędzy obu pozycjami. Zasadniczą wspólną częścią są rozdziały opisujące języki Concurrent Pascal i Modula, zakres pozostałych rozdziałów natomiast, bądź sposób ujęcia tematyki są zdecydowanie różne. Przykładowo, niemal połowa książki jest poświęcona dokładnemu omówieniu mechanizmów synchronizacji procesów w środowisku zwartym, co w skrypcie jest ujęte znacznie zwięźlej. Natomiast w skrypcie dokładniej omówiono mechanizmy komunikacji procesów w środowisku rozproszonym, opisano języki Ada i Chill oraz omówiono zagadnienia programowania sterowanego przepływem danych.

1. WPROWADZENIE

1.1. Pojęcia podstawowe

Tradycyjne, von neumannowskie systemy komputerowe wykonywały jedną operację w danym momencie czasu. Tymczasem już od dawna dostrzegano możliwość innej organizacji pracy maszyn liczących tak, aby dzięki jednoczesnemu wykonywaniu wielu operacji uzyskać przyspieszenie obliczeń. Już w 1842 r. Charles Babbage - znany pionier na polu konstrukcji maszyn liczących - pisał, że "... gdy zachodzi potrzeba wykonania długiej serii identycznych obliczeń, takich jak przy tworzeniu tablic numerycznych, maszynę można skonstruować tak, by produkowała jednocześnie kilka wyników w tym samym czasie, co znacznie skróci cały proces obliczeń".

W powojennej historii rozwoju informatyki już pod koniec lat czterdziestych powstał pierwszy system dwuprocessorowy, w latach pięćdziesiątych zaś projekty specjalizowanych procesorów wielooperacyjnych (np. do rozwiązywania cząstkowych równań różniczkowych). Zasadniczych źródeł nowej organizacji komputerów należy jednak dopatrywać się w pracach z początku lat sześćdziesiątych, kiedy powstały wieloprogramowe systemy komputerowe.

Wśród obecnych systemów komputerowych umożliwiających jednoczesne wykonywanie wielu czynności można wyróżnić trzy powszechnie spotykane typy. Będziemy je nazywać systemami równoległymi lub współbieżnymi. Pierwszy stanowią systemy wieloprogramowe i wielodostępne z podziałem czasu, które dzięki jednoczesnemu wykonywaniu wielu zadań efektywnie wykorzystują swe zasoby obliczeniowe. Drugim są systemy działające w czasie rzeczywistym, np. systemy rezerwacji biletów lotniczych, systemy bankowo-rozliczeniowe, systemy sterowania procesami technologicznymi. Wreszcie typ trzeci to dużej wydajności komputery wieloprocessorowe (np. procesory wektorowe, macierzowe).

Pierwszymi programistami, którzy zetknęli się ze zjawiskiem równoczesności wielu operacji (musieli je opanować i wykorzystać), byli programiści systemowi - ci, którzy tworzyli oprogramowanie podstawowe (systemy operacyjne) komputerów. Z ich potrzeb wynika konieczność posiadania abstrakcyjnego (niezależnego od konkretnego komputera) narzędzia programowania. Łatwo i bardzo wcześnie przekonali się oni o tym, że programując w języku maszynowym narażają się na niezwykle uciążliwy i wrażliwy na błędy proces projektowania. Problemy rozdziału zasobów, szeregowanie zadań, zabezpieczenie równoległe wykonywanych zadań przed

interferencją okazały się bardzo trudne do opanowania bez pomocy języka wysokiego poziomu (maszynowo niezależnego).

Koniec lat sześćdziesiątych można uważać za okres narodzin pierwszych koncepcji abstrakcyjnego spojrzenia na systemy współbieżne. Najważniejszym pojęciem, które się wtedy wykluko jest pojęcie procesu. Przez proces rozumie się sekwencję akcji, która towarzyszy wykonywaniu programu sekwencyjnego; akcje są traktowane abstrakcyjnie, tzn. niezależnie od fizycznych cech procesora oraz od realnego czasu wymaganego na ich wykonanie. Szczegółowe definicje procesu używane w literaturze są bardzo różnorodne. W tym miejscu poprzestaniemy na tym ogólnym określeniu, natomiast w dalszej części skryptu będzie ono odpowiednio precyzowane.

Pojęcie procesu będzie w skrypcie używane w podwójnej roli: w roli procesu abstrakcyjnego - tak jak określono to powyżej - oraz w roli procesu jako jednostki strukturalizacji programów współbieżnych. Kontekst, w którym pojęcie to wystąpi, będzie jednoznacznie wyznaczać jego sens.

Jako przykład roli procesów można rozważyć system wieloprogramowy realizujący pewną liczbę zadań różnych, niezależnych użytkowników. Każde zadanie użytkowe (program sekwencyjny) można utożsamić z pewnym procesem, podobnie można jako procesy traktować systemowe procedury obsługi autonomicznie działających urządzeń wejścia/wyjścia. W czasie działania systemu procesor i pamięć operacyjna mają być na przemian udostępniane różnym procesom w taki sposób, że wszystkie procesy "posuwają się naprzód", mimo że żaden z nich nie ma wyłącznego dostępu do wszystkich wymaganych zasobów.

Tradycyjne języki programowania, takie jak Fortran, Algol 60, Pascal, służą do opisu pojedynczych procesów. Dwa procesy opisane przez dwa programy w takich językach są niezależne i nie mogą współdziałać ze sobą. Natomiast procesy w systemach współbieżnych mogą oddziaływać na siebie. Oddziaływania procesów wynikają głównie z dwóch powodów. Powód pierwszy - to potrzeba wymiany informacji pomiędzy procesami - komunikacja międzyprocesowa. Powód drugi - to potrzeba zachowania określonej kolejności zdarzeń w procesach - synchronizacja procesów.

Konieczność synchronizacji wynika z różnych przyczyn, z których dwie są szczególnie powszechne. Pierwsza - wynika z nałożenia na procesy pewnej relacji poprzedzania, określającej kolejność realizacji procesów. Przykładowo, jeżeli dwa procesy prowadzą komunikację pomiędzy sobą w taki sposób, że jeden proces - producent dostarcza pewne dane poprzez wspólny bufor drugiemu procesowi - konsumentowi, to producent nie może produkować (prowadzić obliczeń) więcej danych dopóty, dopóki konsument nie zwolni bufora z wcześniej wyprodukowanych danych. Druga - wynika z potrzeby systematycznego korzystania ze wspólnych zasobów proce-

sów. Przykładowo przydział zasobów może odbywać się na zasadzie wzajemnego wykluczenia, zgodnie z którą proces uzyskuje wyłączny dostęp do zasobu pod warunkiem, że zasób ten zostanie zwolniony przez inne procesy. Według bardziej złożonych zasad dopuszcza się jednocześnie wykorzystanie zasobu przez większą liczbę procesów, przy czym przydziału dokonuje się na podstawie priorytetów.

Tradycyjne języki programowania nie mają możliwości wyrażania tego rodzaju oddziaływań między procesami. Aby wyrazić takie oddziaływania, programy napisane w języku tradycyjnym musiałyby odwoływać się do procedur systemu operacyjnego nadzorującego wykonywanie tych programów.

Poza brakiem możliwości wyrażania interakcji międzyprocesowej, tradycyjne języki nie mają także możliwości wyrażenia obliczeń niedeterministycznych. Chodzi tu o niedeterminizm wynikający z uzależnienia wyników programu od czasu pojawiania się w systemie pewnych zdarzeń, w szczególności od czasu realizacji poszczególnych fragmentów programu. Przykładowo, jeżeli w systemie rezerwacji biletów lotniczych dwóch klientów z dwóch niezależnych terminali zażąda rezerwacji ostatniego biletu na ten sam lot, to tylko jednemu z nich zostanie przyznana rezerwacja, przy czym nie jest zdeterminowane, któremu z nich.

1.2. Kierunki rozwoju

Aktualny stan rozwoju języków programowania współbieżnego charakteryzują trzy zasadnicze kierunki bazujące na różnych metodach obliczeń równoległych:

- procesy równoległe w zwartym środowisku,
- procesy równoległe w rozproszonym środowisku,
- obliczenia synchronizowane przepływem danych.

Dwa pierwsze podejścia (imperatywne) można traktować jako rozwinięcie i uogólnienie koncepcji von Neumanna. Podejście trzecie (funkcjonalne) należy natomiast uważać za podejście alternatywne.

Pierwszy model procesów równoległych w zwartym środowisku jest historycznie najstarszy. Jest on modelem klasycznego systemu wieloprogramowego. Zakłada się, że wykonywane procesy mają dostęp do wspólnych zasobów - do wspólnej pamięci. Oznacza to, że komunikacja pomiędzy procesami odbywa się wyłącznie za pośrednictwem wspólnych buforów. Pierwszą konstrukcją językową służącą do wyrażania takiego sposobu prowadzenia obliczeń były tzw. semaforey Dijkstry, powstałe w 1965 r. i zastosowane po raz pierwszy w projekcie systemu operacyjnego T.H.E. w 1968 r. [19, 20]. Drugim przykładem konstrukcji są tzw. monitory Hoare'a z 1972 r., które najefektywniejsze zastosowanie znalazły w języku Concurrent Pascal zaprojektowanym przez Brinch Hansena w 1975 r. [7,8,9,33,34]. Języ-

kami programowania, które należą do tego kierunku oprócz Concurrent Pascala są przykładowo: Algol [68,] CCNPascal [52], Iliad [60], Modula [67], Pascal Plus [64], Loglan [59], Edison *).

Drugi model procesów równoległych w rozproszonym środowisku łączy się przede wszystkim z systemami wieloprocesorowymi. Jednym ze sposobów organizacji pracy takich systemów może być przydział oddzielnego procesora dla pojedynczego procesu. Każdy procesor ma własną pamięć, która jest dostępna wyłącznie wykonywanemu na nim procesowi. Procesory są połączone pomiędzy sobą pewnymi kanałami komunikacyjnymi, przez które mogą przekazywać sobie informację - przesyłać wiadomości - i jest to jedyna forma komunikacji. Najważniejsze koncepcje językowe, jakie stworzono w tym zakresie, wiążą się z pracami Brinch Hansena [6,11], Hoare'a [35], Kahna [41,42], Hewitta [29]. Przykładami języków należących do tej grupy mogą być: PLits [24], FORTRAN 77 [66], IVTRAN, LRITRAN, ASC-FORTRAN, CFD, STARAN, TRANQUIL [45] i wreszcie najnowszy, najbardziej okazały przedstawiciel tej grupy - język Ada [40].

Oprócz języków, które bazują w całości na jednym z wymienionych modeli, istnieją także języki przyjmujące mieszany model środowiska obliczeniowego (model mający cechy środowiska zwartego i rozproszonego). Najbardziej znaczącym przedstawicielem tej grupy jest język Chill [71].

Trzeci model obliczeń jest całkowicie odmienny od dwóch poprzednich. Języki programowania poprzednich grup można traktować jako pewne rozszerzenie języków sekwencyjnych poprzez dołączenie do nich mechanizmów obliczeń równoległych. Oznacza to, że program składa się z pewnych fragmentów - procesów, które w uproszczeniu można traktować jako pewne programy sekwencyjne. Model obliczeń synchronizowanych przepływem danych (data flow computations) traktuje program jako nieuporządkowany zbiór instrukcji (tzn. tekst programu nie wyznacza kolejności realizacji instrukcji) i zakłada, że wykonanie danej instrukcji następuje w dowolnym momencie, gdy są gotowe (są wyliczone) argumenty tej instrukcji. Oznacza to, że nie są wprowadzone tutaj pojęcia procesu (abstrakcyjnie każdą instrukcję można uważać za proces, który powstaje w chwili rozpoczęcia jej obliczania i ginie po zakończeniu obliczenia). Taka zasada obliczeń pozwala potencjalnie na najwyższy stopień równoległości obliczeń. Za głównego inicjatora prac nad tymi obliczeniami uważa się Dennisa i jego współpracowników z MIT (Massachusetts Institute of Technology), których prace, m.in. [13,16,17,18], zapoczątkowały nowy kierunek badań. Dotychczasowe prace mają głównie charakter teoretyczny. Towarzyszą im rozważania nad nową architekturą systemów komputerowych, gdyż obecne

*) P. BRINCH HANSEN, Edison - a multiprocessor language, Softw.Fract. Exp., vol. 11, 325-328, 1981.

architektury nie pozwalają na efektywną implementację języków obliczeń synchronizowanych przepływem danych.

Poza omawianymi obliczeniami sterowanymi przepływem danych, które - jak się wydaje - są obecnie najważniejsze spośród koncepcji obliczeń funkcyjnych, były podejmowane próby prezentacji i rozwijania innych nie von neumannowskich modeli obliczeń. Przykładem mogą tu być koncepcje maszyn rekurencyjnych Głuszkowa, maszyn sterowanych rekurencyjnymi strukturami danych i inne^{*)}.

Oprócz omówionych kierunków rozwoju języków programowania współbieżnego należy jeszcze wyróżnić główne kierunki zastosowań. Można tu wymienić trzy częściowo pokrywające się obszary, określane umownie jako programowanie:

- równoległe,
- współbieżne,
- rozproszone.

Jakkolwiek wprowadzone tutaj pojęcia nie są na ogół wyraźnie precyzowane, to jednak wyróżnia je pewien kontekst, w którym są używane. Kolejność, w jakiej pojęcia te wymieniono, odpowiada coraz szerszemu stopniowi ich ogólności.

Programowanie równoległe (parallel programming) jest skojarzone z obliczeniami, które charakteryzują się tym, że wynik obliczeń zależy wyłącznie od danych początkowych (brak niedeterminizmu w zagadnieniu). Dobrym przykładem mogą być obliczenia numeryczne, np. szukanie pierwiastków wielomianu, rozwiązywanie układu równań algebraicznych, znajdowanie ekstremów funkcji itp. Oznacza to, że "natura" problemów jest taka, iż można je rozwiązywać zarówno metodami programowania sekwencyjnego, jak i równoległego.

Programowanie współbieżne (concurrent programming) kojarzy się z obliczeniami, których wyniki są dodatkowo uwarunkowane zdarzeniami zewnętrznymi, jakie mogą zachodzić podczas obliczeń. Klasycznym przykładem są obliczenia wykonywane przez systemy operacyjne systemów wielodostępnych.

Programowanie rozproszone (distributed programming) ma wszystkie atrybuty programowania współbieżnego wraz z dodatkowym założeniem, że komunikacja pomiędzy oddzielnymi, przestrzennie oddalonymi procesorami jest prowadzona za pośrednictwem środków łączności wprowadzających wiele dodatkowych ograniczeń. Ograniczenia te, z których najistotniejsze są "znaczące" opóźnienie i "znacząca" zawodność w przekazywaniu informacji, komplikują komunikację pomiędzy procesami, prowadząc do pojęcia

^{*)} E.V. KOTOV, On basic parallel language, Information Processing 80, S.H. Lavington (ed.), North Holland Publishing Company, 229-240, 1980.

tw. protokołów komunikacji międzyprocesowej. Protokoły są zasadami prowadzenia komunikacji mającymi eliminować, lub co najmniej ograniczać, skutki zawodnego działania pośredniczących środków łączności. Dobry przykład obliczeń rozproszonych stanowią obliczenia rozproszonego systemu operacyjnego sieci komputerowej. (Rozproszony system operacyjny jest realizowany przez zbiór współdziałających systemów operacyjnych poszczególnych komputerów wchodzących w skład sieci.)

1.3. Źródła równoległości

Źródła równoległości obliczeń należy poszukiwać w systemach komputerowych oraz w obliczanych problemach.

Wieloprocessory i sieci komputerowe

Rozwój systemów komputerowych charakteryzują dwie tendencje: coraz większy stopień miniaturyzacji sprzętu liczącego oraz coraz większy stopień terytorialnego rozproszenia połączonego wspólnymi środkami komunikacji. Pierwsza z tych tendencji uwidacznia się w konstrukcji wieloprocessorów wytwarzanych w technologii układów wielkoskalonych. Druga tendencja wyraża się w tworzeniu sieci komputerowych. Pomiedzy tymi tendencjami skrajnymi istnieje większa gama rozwiązań systemów wieloprocessorowych [23]. Komórkowa konstrukcja wieloprocessorów (procesory macierzowe) w szczególny sposób uprzywilejowuje działania na macierzach, umożliwiając znaczne przyspieszenie np. operacji algebraicznych na macierzach [45,66]. Dużą liczbę takich operacji spotyka się przy rozwiązywaniu układów równań różniczkowych, stąd zastosowanie specjalizowanych procesorów macierzowych (np. ILLIAC IV, CDC STAR) w obliczeniach energii atomowej, symulacji sytuacji meteorologicznej opartej na danych pobieranych z sieci stacji obserwacyjnych itp. Ogólnie wieloprocessory stwarzają jednak wiele ograniczeń, które - jak dotychczas - powodują, że związane z nimi języki programowania są przeznaczone przede wszystkim do obliczeń numerycznych. Takie języki, jak np. IVTRAN, IRLTRAN, ASCFORTRAN są zbudowane na bazie FORTRANu z dodatkowym repertuarem działań na macierzach oraz mechanizmami wskazywania działań równoległych. Na przykład fragment programu w języku IVTRAN:

```
DO 1 FOR ALL (I,J,K)/[1...3] x [1...7] x [1...10]
  IF(A(I,J,K).LT.O.O)A(I,J,K) = -A(I,J,K)
1 A(I,J,K) = SQRT(A(I,J,K))
```

oznacza, aby jednocześnie dla wszystkich elementów tablicy $A(I,J,K)$ o rozmiarach $3 \times 7 \times 10$ obliczyć kwadratowe pierwiastki z modułów jej elementów. Etykieta 1 po słowie DO wskazuje koniec "czoła" cyklu.

Sieci komputerowe stwarzają także możliwości równoległego przetwarzania programów. Nie narzucają one takich ograniczeń jak wieloprocesory, dając możliwości równoległego obliczania zagadnień dowolnych typów. Obecnie jednak trudno wskazać implementację języka programowania równoległego poziomu algorytmicznego w sieci komputerowej o architekturze otwartej^{*)}. Przykładami takich języków w specjalizowanych sieciach komputerowych są języki PLITS [24] oraz ILIAD [60].

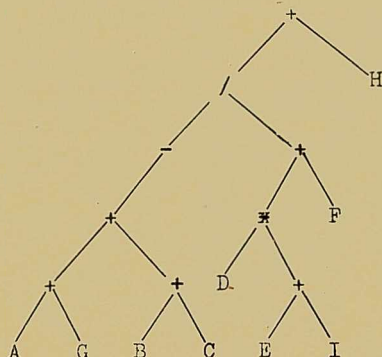
Równoległość w programach sekwencyjnych

Dokładna analiza programów sekwencyjnych, napisanych w takich językach jak Pascal, Algol, Fortran, pozwala stwierdzić, że ich wykonanie nie musi zawsze przebiegać dokładnie sekwencyjnie. Istnieje wiele algorytmów rozpoznawania równoległości w programach sekwencyjnych i zamiany programów sekwencyjnych na równoległe [46,47,55,63].

Najbardziej elementarnymi konstrukcjami języka programowania sekwencyjnego, w których tkwią źródła równoległości, są wyrażenia stanowiące element instrukcji podstawienia, instrukcji warunkowych itd. Wyrażenia takie w obliczeniach sekwencyjnych są obliczane za pomocą odpowiedniego ciągu operacji. W wielu wypadkach obliczenie to można przyspieszyć przez jednoczesne obliczenie kilku operacji. Przykładowo wyrażenie

$$e = - (A + G + B * C) / (D * (E + I) + F) + H$$

ma drzewo rozbioru postaci:



Oznacza to, że mając do dyspozycji równoległe pracujące procesory, możemy obliczenie wyrażenia rozpocząć od jednoczesnego obliczenia wartości wyrażeń:

^{*)} J.A. BARCHAŃSKI, Sieci komputerowe, WNT (w przygotowaniu).

$$e_1 = A + G, \quad e_2 = B * C, \quad e_3 = E + I$$

Dalej, w zależności od tego, które z tych wyrażeń zostaną obliczone wcześniej można wykonywać dalsze obliczenia, np.

$$e_4 = e_1 + e_2 \quad \text{lub} \quad e_5 = D * e_3 \quad \text{itd.}$$

Dopuszczalna kolejność wykonywania działań wynika w oczywisty sposób ze struktury grafu rozbioru.

Druga możliwość zrównoleglenia działań w programie sekwencyjnym wynika z istnienia instrukcji niezależnych. Dwie instrukcje I_1 oraz I_2 są niezależne, jeżeli dla dowolnych danych początkowych ciągi $I_1; I_2$ oraz $I_2; I_1$ produkują te same wyniki końcowe. Jeżeli na przykład dane są dwie kolejne instrukcje

$$x := W_1(a, b);$$

$$y := W_2(b, c);$$

w których x, y, a, b, c są rozłącznymi zmiennymi, W_1 oraz W_2 - wyrażeniami, to takie instrukcje są niezależne i mogą być wykonywane równolegle.

Kolejną możliwość stanowi zrównoleglenie działań iteracyjnych. Rozpatrzmy instrukcję

$$\text{for } i := 1 \text{ to } n \text{ do } A[i] := W(i),$$

w której $W(i)$ jest wyrażeniem zależnym od indeksu i , a niezależnym od elementów $A[j]$ dla $j \neq i$. Jest oczywiste, że w tej sytuacji instrukcje podstawienia $A[i] := W(i)$ dla $i = 1, 2, \dots, n$ można wykonać równolegle.

Algorytmy równoległe

Świadomość możliwości równoległego wykonania programu powinna towarzyszyć programiście w chwili przystąpienia do rozwiązywania problemu. Daje to możliwość odpowiedniego ułożenia algorytmu rozwiązania. Na przykład znany sekwencyjny algorytm obliczania wartości wielomianu opiera się na schemacie Hornera, tzn. wartość wielomianu

$$p(x) = a_n x^4 + \dots + a_1 x + a_0$$

oblicza się wg schematu:

$$p_n = a_n$$

$$p_i = p_{i+1} \cdot x + a_i \quad \text{dla } i = n-1, \dots, 1, 0$$

gdzie $p(x) = p_0$. Czas obliczeń jest oczywiście proporcjonalny do n .

Równoległy algorytm Estrina [51] oblicza wielomian wg schematu:

$$p(x) = q(x) x^{x/2 + 1} + r(x),$$

gdzie

$$q(x) = a_n x^{n/2} + \dots + a_{n/2} + 1,$$

$$r(x) = a_{n/2} x^{n/2} + \dots + a_0,$$

a następne wielomiany $q(x)$, $r(x)$ oblicza się przez podobny binarny podział. Zatem algorytm startuje od jednoczesnego obliczenia

$$a_1 x + q_2, \quad a_3 x + a_2, \dots,$$

a następnie oblicza

$$(a_3 x + a_2) x^2 + (a_1 x + a_0), \dots$$

Jeżeli dostępna jest nieograniczona liczba procesorów, to czas obliczeń jest proporcjonalny do $2 \log n$.

Algorytm Dorna [51] stosuje schemat Hornera k -tego rzędu, który oblicza najpierw równolegle

$$q_0(x^k) = a_0 + a_k x^k + a_{2k} x^{2k} + \dots$$

$$q_1(x^k) = a_1 + a_{k+1} x^k + \dots$$

$$\dots\dots\dots$$

$$q_{k-1}(x^k) = a_{k-1} + a_{2k-1} x^k = \dots,$$

a następnie

$$p(x) = q_0(x^k) + x \cdot q_1(x^k) + \dots + x^{k-1} \cdot q_{k-1}(x^k).$$

Dla k procesorów czas obliczeń szacuje się jako proporcjonalny do $2n/k + 2\log k$.

Innym przykładem algorytmu równoległego jest wyznaczenie **komponentów spójności** grafu [32]. Niech $V = \{0, 1, \dots, n-1\}$ będzie zbiorem wierzchołków grafu nieskierowanego $G = (V, E)$, gdzie $E \subset V \times V$ jest zbiorem krawędzi. Przedstawmy graf w postaci macierzy boolowskiej A o wymiarach $n \times n$, gdzie $A(i, j) = \text{true}$ wtedy i tylko wtedy, gdy $(i, j) \in E$. Komponentą spójności grafu G jest największy podgraf G_c taki, że pomiędzy dowolnymi wierzchołkami G_c istnieje łączący je ciąg krawędzi. Każdy wierzchołek należy dokładnie do jednej komponenty spójności. Niech m -wymiarowy wektor D służy do wyznaczenia komponent: jeżeli $G_c = (V_c, E_c)$ jest komponentą spójności, to dla $i \in V_c$ $D(i)$ jest równe najmniejszemu elementowi z V_c . Algorytm ma postać następującą:

1. Dla wszystkich i wykonuj $D(i) := i$. Powtarzać następnie kroki od 2 do 6, przez co najwyżej $\log n$ iteracji.
2. Dla wszystkich i wykonuj

$$C(i) = \begin{cases} \min \{ D(j) \mid A(i, j) = \text{true} \wedge D(j) \neq D(i) \} \\ D(i) \quad \text{gdy podany wyżej zbiór jest pusty.} \end{cases}$$

3. Dla wszystkich i wykonuj

$$C(i) = \begin{cases} \min \{ C(j) \mid D(j) = i \wedge C(j) = i \} \\ D(i) \text{ gdy podany wyżej zbiór jest pusty.} \end{cases}$$

4. Dla wszystkich i wykonuj $D(i) := C(i)$.

5. Przez co najwyżej $\log n$ iteracji dla wszystkich i wykonuj $C(i) := C(C(i))$.

6. Dla wszystkich i wykonuj

$$D(i) := \min(C(i), D(C(i))).$$

Działanie algorytmu jest następujące. Podczas pierwszej iteracji są badane wierzchołki przyległe do każdego wierzchołka (kroki 2,3) oraz zbiory wierzchołków, które połączone są ze sobą. W rezultacie każdy taki zbiór wierzchołków jest zanurzony w pewnym nadzbiorze zdefiniowanym przez D następująco: $D(i)$ jest równe wierzchołkowi o najmniejszym numerze, do którego przylega wierzchołek i . W kolejnych iteracjach są badane krawędzie łączące dwa sąsiednie nadzbiory (kroki 2,3), a następnie nadzbiory te są łączone ze sobą w krokach 4-6. Obliczenia kontynuuje się aż do chwili, gdy wszystkie nadzbiory zostaną połączone, a dalsze iteracje nie wnoszą żadnych zmian. Okazuje się, że liczba iteracji nie przekracza $\log n$.

Wybrane algorytmy są tylko ilustracją równoległego podejścia na etapie rozwiązania problemu. Możliwości, jakie kryją się w tego rodzaju podejściu, pozwalają oczekiwać, że przed klasycznymi, już rozwiązanymi, zagadnieniami, w szczególności przed metodami numerycznymi, stoi powtórna droga do przebycia.

ĆWICZENIA

1. Przytoczone opowiadanie [37] stanowi pouczający przykład ilustrujący zagadnienie synchronizacji dostępu do wspólnych zasobów.

Wysoko w Andach przebiegają dwie okólne linie kolejowe, z których jedna należy do Boliwii, a druga do Peru. Linie te mają wspólny odcinek toru prowadzący przez przełęcz górską leżącą na granicy obu państw. Przejazd przez ten odcinek odbywa się zawsze tylko w jednym, na stałe przyjętym, kierunku. Problem polega na tym, że maszyniści wjeżdżający na ten odcinek są ślepi i głusi. Aby uniknąć kolizji uzgodniono następujące reguły wjazdu na wspólny tor. Przed wjazdem umieszczono beczkę. Maszynista zanim wjedzie na wspólny tor zatrzymuje pociąg, podchodzi do beczki i sprawdza jej zawartość. Jeżeli beczka jest pusta, to wyszukuje kamień i wrzuca do go beczki, wskazując tym, że zajmuje tor, wsiada do pociągu i przejeżdża wspólny odcinek. Po przejeździe odcinka ponownie wraca do beczki i wyrzuca z niej kamień, wskazując zwolnienie wspólnego toru. Jeżeli przed wjazdem maszynista znajdzie w beczce kamień, to robi przerwę na krótką sjeść, po czym ponownie, często kilkakrotnie, spraw-

dza czy wreszcie beczka jest pusta. Pewien spozstrzegawczy kolejarz boliwijski stwierdził, że wywrotowy peruwiański rozkład jazdy spowoduje stałą blokadę boliwijskich pociągów. (Dlaczego?) Na to boliwijski maszynista stwierdził, że nigdy to się nie zdarzy. (Dlaczego?) Niestety pewnego dnia pociągi na przejeździe zderzyły się. (Dlaczego?). Po katastrofie spozstrzegawczy kolejarz zaproponował nowe rozwiązanie. Boliwijski maszynista powinien czekać przy beczce dopóty, dopóki beczka będzie pusta, a po przejeździe wspólnego odcinka powinien wrócić do beczki i wrzucić do niej kamień. Natomiast maszynista peruwiański powinien czekać przy beczce, aż w beczce będzie kamień, a po przejeździe toru powinien beczkę opróżnić. Kolejarz był przekonany o bezkolizyjności swego rozwiązania. Jednak peruwiańczycy okazali się bardzo nieszcześliwi z powodu nowej umowy. (Dlaczego?) W związku z tym spozstrzegawczy kolejarz zaproponował kolejne rozwiązanie. Przed wjazdem umieszczono dwie beczki, każdą dla jednego maszynisty. Gdy maszynista osiąga przejazd podchodzi do swej beczki, wrzuca do niej kamień, a następnie sprawdza drugą beczkę. Jeżeli jest pusta, to wsiada do pociągu, przejeżdża wspólny tor, wraca do swej beczki i opróżnia ją. Jeżeli w beczce jest kamień, to wraca do swej beczki, wyrzuca z niej kamień i robi krótką przerwę na sjęstę, po której ponawia poprzednie czynności, zaczynając od wrzucenia kamienia do swej beczki. Metoda działała przez wiele miesięcy, aż pewnego dnia późną jesienią pociągi zostały zablokowane na przeciąg wielu sjęst. (Dlaczego?)

2. Podaj zasady szacowania minimalnej liczby procesorów niezbędnych do możliwie szybkiego obliczenia danych wyrażeń liczbowych:

- przy założeniu jednakowych czasów wykonywania różnych operacji,
- przy założeniu dowolnych (znanych) czasów wykonywania operacji.

3. Przedstaw optymalny czasowo harmonogram obliczeń wyrażenia

$$(a + b) / (a - b) * (c/e - c*d) * (a*a + c*d) / ((a+b)*(a-b))$$

przy założeniu, że czas wykonywania operacji +, - wynosi T, * wynosi 3T, / wynosi 4T.

4. Przedstaw potencjalne równoległości w ciągu instrukcji

for i:=1 to m do

for j:=1 to n do

begin

A[j] := X[i] + Y[j];

B[i,j] := X[i] * i;

C[k,j] := A[k];

D[i,k,j] := Y[i] * k * (i-j)

and;

5. Zaproponuj równoległy algorytm obliczenia zera funkcji z wykorzystaniem metody Newtona.

6. Zaproponuj algorytm równoległego sortowania liczb

- przy dowolnej liczbie procesorów
- przy ograniczonej liczbie procesorów.

7. Zaproponuj algorytm równoległy wyszukiwania najkrótszej ścieżki w grafie skierowanym z łukami obciążonymi liczbami nieujemnymi.

8. Dana jest sieć procesorów połączonych w dwuwymiarową strukturę macierzową. Zaproponuj zasady komunikacji między sąsiednimi procesorami. Uwzględnij te zasady do obliczenia iloczynu macierzy liczbowych.

2. PORÓWNANIE PODSTAWOWYCH KONCEPCJI

2.1. Zasady porównania

Ocena i porównanie (pragmatyka) koncepcji programowych nie jest zagadnieniem łatwym [22]. Należy bowiem brać pod uwagę wiele porównywalnych aspektów. Najbardziej istotnymi aspektami są: siła ekspresji języka, jego czytelność i łatwość implementacji.

Siła ekspresji języka programowania współbieżnego wyraża rodzaje równoległości, jakie dopuszcza język, sposoby komunikacji i synchronizacji, rodzaje niedeterminizmu. Prawidłowo zaprojektowany język powinien zawierać dostatecznie bogaty zestaw tych mechanizmów, tak by programista mógł opisywać w nim wszystkie przewidywane akcje, bez odwoływania się do procedur systemu operacyjnego.

Czytelność programów napisanych w danym języku określa stopień trudności analizy programu (określenie tego, co program wykonuje) na podstawie analizy tekstu programu (a nie analizy wykonania programu). Zasadniczym wymogiem czytelności programu jest jego strukturalizacja (modularyzacja), tzn. taka dekompozycja programu na dostatecznie małe fragmenty, które można analizować oddzielnie, niezależnie od pozostałej części programu. Dekompozycja jest krytyczną własnością w projektowaniu programów współbieżnych, gdyż pomiędzy komponentami programu zachodzą subtelne oddziaływania i mogą one doprowadzić do blokad, hazardów lub innych form nieprawidłowych zachowań. Czytelność programów jest ważna zarówno przy wszelkich modyfikacjach lub rozbudowie programu, jak również podczas jego konstrukcji: język dysponujący konstrukcjami wysokiego poziomu kojarzącymi mechanizmy komunikacji i synchronizacji nie tylko przyspiesza analizę i syntezę programów, lecz także czyni programy bardziej odpornymi na błędy.

Względy implementacyjne odnoszą się do procesu tworzenia kompilatora języka oraz do efektywności pracy tego kompilatora (efektywność przekładu i efektywność działania kodu wynikowego). Na ogół względy te pozostają w sprzeczności z poprzednio wymienionymi aspektami: wzrost siły ekspresji lub podniesienie czytelności pociąga dodatkowe kłopoty implementacyjne.

Porównanie przedstawionych dalej koncepcji programowych będzie polegać na ocenie omówionych aspektów.

2.2. Przykładowe systemy

Prezentacji i porównania podstawowych koncepcji programowych dokonano w sposób poglądowy na podstawie następujących dwóch przykładów.

P.1. System rezerwacji biletów lotniczych składa się ze wspólnej bazy danych, do której w trybie interakcyjnym ma dostęp pewna liczba zdalnych użytkowników. System wymaga współbieżnego działania w celu zapewnienia podziału dostępu do wspólnych danych, obsługi napływających zgłoszeń klientów, sensownej przepustowości.

Przyjmijmy, że wspólna baza danych zawiera informacje o pewnej liczbie lotów. W uproszczeniu, niech każdy lot dysponuje 100 miejscami dla pasażerów. Aby zarezerwować m miejsc w locie numer 1, użytkownik wysyła wiadomość (polecenie)

('rezerwuj', 1, m).

Jeżeli dany lot dysponuje wymaganą liczbą miejsc, to dokonuje się rezerwacji i informuje zwrotnie użytkownika, w razie braku odpowiedniej liczby miejsc, rezerwacja nie jest realizowana, a użytkownik otrzymuje odpowiedź negatywną. Użytkownik może także sprawdzić iloma wolnymi miejscami dysponuje dany lot; w tym celu wysyła on wiadomość

('informuj', 1),

a w odpowiedzi system przesyła mu aktualną liczbę wolnych miejsc.

P.2. System buforowania składa się z pewnej liczby urządzeń wejściowych, z których czytane są pojedyncze porcje danych do wspólnego bufora, w których dane te są umieszczane oraz z pewnej liczby urządzeń wyjściowych, które pobierają ze wspólnego bufora pojedyncze porcje danych i wyprowadzają na zewnątrz. Urządzenia wejściowe (czytniki) oraz urządzenia wyjściowe (pisaki) pracują współbieżnie i są tylko uzależnione od dostępu do wspólnego bufora (czytniki - od możliwości zapisu w buforze, pisaki - od możliwości odczytu z bufora). Bufor ma ograniczoną pojemność i dopuszcza się, że w określonym czasie może być dostępny tylko dla jednego czytnika lub pisaka. Aby zapewnić efektywną pracę systemu zakłada się, że operacje zapisu/odczytu do/z bufora są znacznie krótsze od operacji wejścia/wyjścia wykonywanych przez czytniki/pisaki.

Opisane przykłady stanowią tylko przykłady ilustracyjne. Dokonano w nich uproszczeń, pominięto detale. Przykładowo pominięto kwestię działania systemów w warunkach awaryjnych, kwestię zakończenia ich działania. Trudno zatem sformułować niepodważalne wartości o właściwościach języka

programowania na podstawie tak uproszczonych przykładów. Należy także pamiętać o trudnościach, jakie wyniknęłyby z dołączenia urealnających szczegółów.

W bieżącym rozdziale zastosowano pascalopodobną, samowyjaśniającą się notację programową.

2.3. Procesy w środowisku zwartym

Najwcześniejsze podejścia do programowania równoległego przyjmowały podejście naturalnie nasuwające się z obserwacji działania systemów wieloprogramowych. Systemy takie składały się z jednego lub kilku procesorów mających dostęp do wspólnej pamięci. Komunikacja procesów realizowanych w systemie odbywała się za pośrednictwem wspólnej pamięci stanowiącej wspólnę, zwarte środowisko dla tych procesów. Komunikacja wyrażała się przez realizację instrukcji przypisania operujących na zmiennych globalnych reprezentujących obszary wspólnej pamięci.

Współpraca współbieżnych procesów wymaga mechanizmów synchronizacji dostępu do wspólnej pamięci. (Na poziomie architektury sprzętu liczącego jest to realizowane przez wykorzystanie mechanizmu przerwań.) Powstało wiele różnych mechanizmów synchronizacji [12,45]:

- różne odmiany semaforów (Dijkstra 1968, Vontilborgh i van Lamsweerde 1972, Agerwala 1977),
 - warunkowe regiony krytyczne (Hoare 1972),
 - wyrażenia ścieżkowe (Campbell, Habermann 1974),
 - monitory (Hoare 1974, Brinch Hansen 1973),
 - wyrażenia warunkowe (Kessels 1977),
- i inne.

Spośród wielu proponowanych mechanizmów dalej omówiono tylko dwa: semafor Dijkstry oraz monitory Hoare'a (koncepcja) i Brinch Hansena (notacja językowa). Odegrały one chyba najistotniejszą rolę w procesie formowania mechanizmów synchronizacji. Należą one do dwóch różnych kategorii: semafor stanowi specjalny typ zmiennej, na której można wykonywać tylko określone operacje, monitor natomiast stanowi zdefiniowany przez programistę zbiór procedur, poprzez które proces może działać na wspólnych danych.

2.3.1. Synchronizacja procesów za pomocą semaforów

Semafor S jest zmienną typu całkowitego (integer), której, w momencie inicjalizacji, przypisuje się pewną wartość początkową. Ze zmienną S jest związana kolejka, w której przechowuje się nazwy procesów. Na semaforze można wykonać tylko dwie operacje: wait (S) oraz signal (S). (Dijkstra nazywa je odpowiednio P oraz V.) Jeżeli proces P wykonuje

operację wait (S), następuje zmniejszenie wartości S o wartość 1; jeżeli nowa wartość będzie ujemna, to nazwa procesu P będzie zapisana w kolejce stowarzyszonej z S i wykonanie procesu P zostanie zawieszone; jeżeli nowa wartość S nie jest ujemna, to proces P wykonuje się dalej. Jeżeli proces P wykonuje operację signal (S), to następuje dodanie jedynki do S, po czym proces P kontynuuje swe obliczenia. Jeżeli nowa wartość S okaże się dodatnia, to nazwę jednego z zawieszonych procesów usuwa się z kolejki i wznowia wykonywanie tego procesu. Wybór jednego z zawieszonych procesów jest niedeterministyczny.

Semaforey dostarczają więc środków do zawieszenia wykonywania procesów, aż do momentu spełnienia pewnych warunków. Jeżeli proces wykonuje operacje na semaforze w połączeniu z dostępem do zmiennych globalnych, można osiągnąć prawidłową synchronizację dostępu. Przykładowo, semafor z początkową wartością 1 można zastosować do zachowania wzajemnie wykluczającego się dostępu do wspólnych zmiennych (można zorganizować sekcję krytyczną).

P.1. Przykład procesu w systemie rezerwacji biletów lotniczych, który usiłuje dokonać rezerwacji miałyby więc postać:

```

process klient_i-ty;
.....
    wait (mutex);
    if liczba_dostępnych_miejsc[nr_lotu] ≥ n
    then
    begin
        liczba_dostępnych_miejsc[nr_lotu] :=
            liczba_dostępnych_miejsc nr_lotu - n;
        powodzenie := true
    end
    else powodzenie := false;
    signal (mutex)
.....
end process

```

gdzie mutex jest zadeklarowany i zainicjowany wspólnie dla wszystkich procesów klientów przez:

```

semaphore mutex;
mutex := 1;

```

Jeżeli kilku klientów pragnie uzyskać dostęp do bazy danych bez naruszenia jej zawartości, to oczywiście dostęp taki mogą uzyskać jednocześnie, bez potrzeby dodatkowej synchronizacji. Proces, który ma zamiar dowie-

dzieć się o liczbie wolnych miejsc, może wykonać po prostu instrukcję:

```
n := liczba_wolnych_miejsc[nr_lotu].
```

Oczywiście w realnych systemach rezerwacji klient może zapytywać o większą liczbę informacji, dlatego zwykle istnieje potrzeba synchronizacji nawet w razie dostępu nie modyfikującego wspólnych danych po to, by informacje nie okazywały się nieaktualne już w chwili przekazania ich do klienta.

P.2. W przykładzie systemu buforowania synchronizacja dostępu do wspólnego bufora jest bardziej złożona i wymaga użycia dwóch semaforów.

```
concurrentprogram system_buforowania;
var wspólny_bufor: T;
semaphore pusty, pełny;
pusty := 1; pełny := 0;
  process czytnik_i-ty;    (i = 1,2,...,m)
  var lokalny_bufor : T;
  begin
  while true do
  begin read (urządzej_i-te, lokalny_bufor)
    wait (pusty);
    wspólny_bufor := lokalny_bufor
    signal (pełny)
  end
  endprocess;
  process pisak k-ty;      (k = 1,2,...,n)
  var lokalny_bufor : T;
  while true do
  begin wait (pełny);
    lokalny_bufor := wspólny_bufor;
    signal (pusty)
    write (urządźwyj_k-te, lokalny_bufor);
  end
  endprocess;
endprogram
```

Semafor "pusty" gwarantuje, że tylko jeden z procesów czytników otrzyma dostęp do wspólnego bufora, podczas gdy semafor "pełny" gwarantuje to samo dla procesów pisaków.

Konstrukcja semafora jest wystarczająca do rozwiązywania szerokiej klasy problemów synchronizacji, chociaż czasem stwarza to duże kłopoty. Nie uwzględnia natomiast semafor dwóch elementów:

- czasu, w którym zachodzi dane zdarzenie: przykładowo nie można wdrożyć zawieszenia się procesu na określony odcinek czasu;
- możliwości wpływu jednego procesu na zakończenia działania innego procesu. Oba te elementy pominięto świadomie, gdyż proces abstrakcyjny nie ma bezwzględny (substancjalny) czas zachodzenia zdarzeń, a jedynym właściwym oddziaływaniem pomiędzy procesami jest przekazywanie danych przez wspólną pamięć.

Pojęcie semafora nie jest jednak wystarczające do rozwiązania zagadnień synchronizacji, przy braku wspólnego, globalnego środowiska. Nie jest możliwy jawny opis oddziaływań pomiędzy procesem komunikującym się przez wspólne łącze.

Koncepcja semafora stanowiła w swoim czasie milowy krok w kierunku lepszego zrozumienia problemów synchronizacji. Jednakże narzędzie to ma znaczne niedogodności. Pierwsza wynika stąd, że jest to narzędzie bardzo elementarne: programiście pozostawia się rozwiązanie sposobu właściwego użycia semaforów. Uzyskanie takiego rozwiązania jest kłopotliwe i bardzo podatne na błędy, a same rozwiązania - zwłaszcza przy złożonych zagadnieniach - są sztywne, trudno modyfikowalne. Druga wada wynika z zupełnego braku modularności. Informacje o tym, w jaki sposób są wykorzystywane wspólne dane, są rozproszone po całym programie. Przykładowo trudno zlokalizować źródła niedeterminizmu w programie.

Brak modularności w połączeniu z elementarnością powoduje bardzo trudną analizę (mała czytelność) programów z semaforami. Natomiast implementacja semaforów nie nastrocza żadnych kłopotów. Historycznym przykładem systemu operacyjnego wykorzystującego semaforów jest system T.H.E. Dijkstry z 1968 r.

2.3.2. Synchronizacja procesów za pomocą monitorów

Monitory wprowadzono przede wszystkim po to, aby uzyskać bardziej strukturalną postać programów współbieżnych. W odróżnieniu od semaforów, pełna informacja o wspólnych zasobach różnych procesów i o sposobie ich wykorzystania jest zawarta w jednym wspólnym fragmencie programu, zwanym monitorem. Deklaracja monitora zawiera definicję wspólnych danych oraz definicję procedur, które mogą na nich działać. Procedury te, nazywane procedurami wejściowymi monitora, są dostępne procesom, tzn. procesy mogą wywoływać jedną z tych procedur w celu wykonania pewnych operacji na wspólnych danych. Na wspólnych danych, wchodzących w skład monitora, nie można wykonywać dowolnych operacji, a tylko te, które są zdefiniowane przez procedury wejściowe. Monitor jest wyłącznie obiektem usługowym, nie może on dokonywać samodzielnie wywołania własnych procedur wejściowych.

Monitor działa w taki sposób, że wykonywanie jego procedur wejściowych wzajemnie się wyklucza. Jeżeli w pewnej chwili nastąpi jednoczesne wywołanie procedur wejściowych przez kilka różnych procesów, to wykona się tylko jedna z tych procedur, dla jednego z tych procesów, podczas gdy pozostałe zostaną zawieszane (wstrzymane), aż do czasu, gdy zakończy się wykonywanie rozpoczętej procedury (zwolnienie monitora przez proces, który poprzednio uzyskał dostęp do monitora). Wybór kolejnego z zawieszonych procesów do wykonania w monitorze procedury wejściowej jest niedeterministyczny.

Proces, który wywoła procedurę wejściową monitora całkowicie poddaje się sterowaniu przez monitor. Proces może poddawać się sterowaniu monitora dwoma sposobami. Pierwszy - prostszy - polega na tym, że od momentu wywołania procedury wejściowej monitora proces ewentualnie oczekuje przez pewien czas na zwolnienie monitora przez inne procesy, następnie jest wykonana żądana procedura wejściowa, a po jej zakończeniu sterowanie zostaje zwrócone procesowi, który może wykonywać swe dalsze instrukcje. Drugi - bardziej złożony - wykorzystuje tzw. zmienne kolejkowe (queue) z operacjami opóźnij (delay) oraz wznów (continue). Zmienne kolejkowe służą do jawnego (programowego) zapamiętywania nazw procesów zawieszanych. Gdy proces P wykona operację opóźnienie na zmiennej kolejkowej q - delay (q), wówczas nastąpi zawieszenie tego procesu, a jego nazwa P zostanie zapamiętana w zmiennej q . Proces P pozostanie zawieszony dopóty, dopóki nie wznowi go inny proces. W momencie zawieszenia procesu monitor jest gotowy do obsługi innych procesów. Gdy proces wykona operację continue (q), wówczas - o ile q nie jest puste - następuje wznowienie procesu, którego nazwa jest aktualnie przechowywana w q .

P.1. Przykład systemu rezerwacji biletów ilustruje prostszy sposób sterowania procesami w monitorze. Monitor zarządzający wspólną bazą danych ma postać:

```

monitor baza_danych
  var liczba_dostępnych_miejsc : array [1..n]
    of integer,
    i : integer;
  procedure entry rezerwuj (lot, liczba : integer,
    var powodzenie : boolean);
  begin
    if liczba_dostępnych_miejsc [lot]  $\geq$  liczba
    then begin
      liczba_dostępnych_miejsc [lot] :=
        liczba_dostępnych_miejsc [lot] - liczba;
      powodzenie := true
    
```

```

    end
    else powodzenie := false
    end;
    procedure entry informuj (lot : integer,
        var liczba : integer);
    begin
        liczba := liczba_dostępnych_miejsc [lot]
    end;
    begin {inicjalizacja monitora}
        for i := 1 to n do
            liczba_dostępnych_miejsc [i] := 100
        end monitor;
    end

```

} ciasto programu

Instrukcja inicjalizacji monitora służy do nadania wartości początkowych zmiennym wspólnym ochranianym przez monitor. Instrukcja powinna być wykonana jednorazowo przed przyjęciem wywołań procedur wejściowych monitora. Wywołania procedur wejściowych w procesach klientów będą miały postać

baza_danych . rezerwuj (L,n,p)

lub

baza_danych . informuj (L,n)

gdzie L,n,p są aktualnymi parametrami wywoływanych procedur.

P.2. Przykład systemu buforowania jest bardziej złożony. Pokazuje on sposób wykorzystania zmiennych kolejkowych do pewnej strategii szeregowania procesów wywołujących monitor (szeregowanie zgodne z kolejnością zgłoszeń). Wewnątrz monitora są zdefiniowane także jego funkcje wewnętrzne, których nie mogą wywoływać procesy, a które mogą być wykorzystane przez procedury wejściowe monitora

```

monitor wspólny_bufor;
var bufor : blok;
    zajęty : boolean;
    kolpis : array [0..n-1] of queue;
    kolczyt : array [0..m-1] of queue;
    pkolpis, kkolpis, pkolczyt, kkolczyt,
    dkolpis, dkolczyt: integer;
function wejście (do : boolean) : integer;
begin
    if do then
    begin
        wejście := kkolpis;
        kkolpis := (kkolpis + 1) mod n;
    end
    end

```



```

    dkolpis := (dkolpis + 1
end
else
begin
    wejście := kkolczyt;
    kkolczyt := (kkolczyt + 1) mod m;
    dkolczyt := (dkolczyt + 1
end
end;
function wyjście (to : boolean) : integer;
begin
    if to then
    begin
        wyjście := pkolpis;
        pkolpis := (pkolpis + 1) mod n;
        dkolpis := dkolpis - 1
    end
    else
    begin
        wyjście := pkolczyt;
        pkolczyt := (pkolczyt + 1) mod m;
        dkolczyt := dkolczyt - 1
    end
end;
function pusta (to : boolean) : boolean;
begin
    if to then pusta := (dkolpis = 0)
    else pusta := (dkolczyt = 0)
end;
procedure entry czytaj (var danewy : blok);
begin
    if not zajęty then
        delay (kolczyt [wejście(false)]);
        danewy := bufer;
        zajęty := false;
    if not pusta (true) then
        continue (kolpis [wyjście(true)])
    end;
procedure entry pisz (danewy : blok),
begin
    if zajęty then
        delay (kolpis [wejście(true)]);

```

```

    bufor := danewe;
    zajęty := true;
    if not pusta (false) then
        continue (kolczyt [wyjście(false)])
    end;
    begin {inicjalizacja monitora}
        zajęty := false;
        pkolpis := 0; kkolpis := 0; dkolpis := 0;
        pkolczyt := 0; kkolczyt := 0; dkolczyt := 0
    end monitor

```

Procesy czytników będą wywoływać procedurę wejściową

wspólny_bufor . pisz (we),

zaś procesy pisaków -

wspólny_bufor . czytaj (wy),

gdzie we, wy są odpowiednimi parametrami aktualnymi procedur ustalonymi przez te procesy.

Siła ekspresji monitorów jest taka sama jak semaforów w tym sensie, że każdy program, który zawiera semaforey może być zastąpiony przez program zawierający monitory oraz odwrotnie.

Konstrukcja monitora jest bardziej modularna niż semafora, dlatego też programy są bardziej czytelne. Sposób dostępu do wspólnych zasobów jest zdefiniowany w jednym tekstowo zwartym module. Czyni to program łatwiejszy do modyfikacji. Przykładowo, gdyby bufor w P.2. składał się z większej liczby bloków, przekonstruowania wymagałby tylko monitor.

Przedstawiona wersja monitora zakłada rozłączne działanie jego procedur wejściowych. Ogólnie tak być nie musi; zwłaszcza wówczas, gdy poszczególne procedury działają na rozłącznych fragmentach wspólnych danych, można założyć równoczesność ich działania [38].

Podobnie jak przy semaforach, implementacja monitorów nie nastrocza większych kłopotów. System operacyjny SOLO Brinch Hansena z 1975 r., napisany w języku Concurrent Pascal, stanowi przykład systemu całkowicie opartego na monitorach.

2.4. Procesy w środowisku rozproszonym

Wraz z rozwojem systemów komputerowych, a zwłaszcza pojawieniem się systemów wieloprocessorowych [23], torowało sobie drogę programowanie oparte na założeniu środowiska rozproszonego. Wprowadzie pojęcia komunikujących się rozproszonych procesów używał Brinch Hansen już w 1970 r. [6], to jednak zasadniczy rozwój takiego programowania nastąpił w poło-

wie lat siedemdziesiątych. Należy wymienić tu prace Kahna [41,42], Hoare'a [35], Brinch Hansena [11], Hewitta [28] i inne, np. [39,48]. Wieloprocesory i sieci komputerowe stanowią naturalną bazę odniesienia dla tego podejścia.

Dla ilustracji wprowadzimy język programowania, w którym zdefiniujemy pojęcie wiadomości (tj. porcji danych przesyłanych pomiędzy procesami) oraz operacji wysyłania i odbioru wiadomości. Przez wiadomość będzie rozumiana trójka

$w = (\text{nadawca}, \text{odbiorca}, \text{treść}),$

gdzie nadawca jest nazwą procesu wysyłającego wiadomość, odbiorca - nazwą procesu odbierającego, treść zaś stanowi przekazywaną informację użytkową. Dla ustalenia uwagi wiadomości będą traktowane jak rekordy (w sensie Pascala) o podanych wyżej nazwach pól, a więc np. wartość nadawca określa nazwę procesu wysyłającego wiadomość w .

Wysyłanie wiadomości w (przez proces P do procesu Q) polega na wykonaniu procedury

send (w)

gdzie

$w.$ nadawca = P ,

$w.$ odbiorca = Q .

Zakłada się, że proces P wysyłający wiadomość w musi ustalić wartość pól odbiorca i treść, natomiast wartość pola nadawca jest ustawiana automatycznie przez procedurę send.

Każdy proces ma jedną kolejkę, w której gromadzi nadchodzące do niego wiadomości. Pobieranie wiadomości z tej kolejki odbywa się przez wywołanie funkcji receive, która powoduje podstawienia pod swą nazwę pierwszej wiadomości z kolejki wejściowej i usunięcie tej wiadomości z kolejki. Jeżeli kolejka jest pusta, wykonanie receive zostaje opóźnione aż do momentu napływu nowej wiadomości.

Omawiane poprzednio przykładowe problemy znajdują tu następujące rozwiązanie:

P.1. W przykładzie systemu rezerwacji biletów, w modelu ze środowiskiem zwartym, zakładano, że procesy klientów mają dostęp do wspólnych danych. Obecnie zakładamy, że tylko jeden proces o nazwie "transakcja" ma taki dostęp, natomiast procesy klientów mają tylko dostęp pośredni poprzez wymianę odpowiednich wiadomości z procesem transakcja.

W systemie są zdefiniowane dwa typy wiadomości: wiadomość od klientów do procesu "transakcja" oraz wiadomości przesyłane w kierunku przeciwnym:

```

message od_klienta =
  record
    nadawca: nazwa_klienta;
    odbiorca: transakcja;
    treść: record
      typ : (rezerwuj, informuj)
      lot , liczba : integer
    end
  end
message do_klienta =
  record
    nadawca: transakcja;
    odbiorca: nazwa_klienta,
    treść: record
      typ : (rezerwuj, informuj)
      powodzenie: boolean;
      liczba : integer
    end
  end

```

Definicja procesu transakcja ma postać następującą:

```

proces transakcja;
var dostępne_miejsca : array [1..n] of integer;
  żądanie: message od_klienta;
  odpowiedź: message do_klienta;
  i,n: integer
begin
  for i := 1 to n do dostępne_miejsce [i] := 100,
  while true do
    begin
      żądanie := receive;
      case żądanie.typ of
        rezerwuj:
          begin
            j:=żądanie.treść.lot,
            i:=żądanie.treść.liczba;
            if dostępne_miejsce [j] ≥ i then
              begin
                odpowiedz.treść.powodzenie:=true;
                dostępne_miejsce [j] :=
                  dostępne_miejsce [j] - i;
              end
            end
          end

```



```

else odpowiedź.treść.powodzenie:=false;
      odpowiedź.treść.typ:=rezerwuj
    end
informuj:
    begin
        odpowiedź.treść.liczba:=
            dostępne_miejsce [żądanie.treść.lot];
        odpowiedź.treść.typ:=informuj
    end
end;
send (odpowieź)
end
endprocess

```

Treść procesów klientów jest tutaj mniej istotna: ważne jest tylko to, że klienci są generatorami wiadomości do procesu "transakcje" oraz odbiorcami odpowiedzi tego procesu.

P.2. W systemie buforowania wyróżnia się trzy rodzaje procesów: n procesów pisaków, m procesów czytników oraz jeden proces sterujący wspólnym buforem. Procesy czytników (pisaków) komunikują się z procesem wspólnego bufora w celu przekazania (odbioru) ustalonych bloków informacji. Procesy czytników po odczytaniu z urządzeń zewnętrznych bloku informacji przekazują je do procesu wspólnego bufora, który z kolei przesyła je do jednego z procesów pisaków. Każdy z procesów pisaków, który zakończy wyprowadzenie bloku informacji wysyła do procesu wspólnego bufora wiadomość o gotowości do przyjęcia nowego bloku danych, dzięki czemu proces wspólnego bufora zna aktualną zajętość pisaków.

W odróżnieniu od poprzedniego przykładu mamy tu bardziej złożoną wymianę informacji pomiędzy procesami. Dlatego wewnątrz procesu wspólnego bufora definiuje się nowy typ danych sequence, służący do formowania w kolejkę napływających wiadomości. Na typie tym można dokonywać operacji wpisu do kolejki inseq, odczytu z kolejki deseq oraz badania zajętości kolejki za pomocą funkcji boolowskiej empty (funkcja przyjmuje wartość true, gdy kolejka jest pusta).

```

concurrentprogram system_buforowania;
    message informacja_uzytkowa =
        record
            nadawca: nazwa_procesu_czytnika_lub_wspolnego_bufora;
            odbiorca: nazwa_procesu_pisaka_lub_wspolnego_bufora;
            treść : blok
        end;

```

```

message informacja_sterujaca =
  record
    nadawca: nazwa_procesu_pisaka;
    odbiorca : wspólny_bufor;
    treść : <pusta>
    {pole treść nie zawiera żadnej informacji
     - dlaczego?}
  end;
process wspólny_bufor;
var koldanych: sequence of message
               informacja_użytkowa;
    kolgotowości: sequence of message
                  informacja_sterujaca,
    wewiad, wywiad : message informacja_użytkowa;
    gotwiad : message informacja_sterujaca
begin
while true do
  begin
    wewiad := receive;
    if wewiad.nadawca ∈ nazwa_procesu_czytnika
    then
      begin
        if empty (kolgotowości) then
          enseq (wewiad.koldanych)
        else
          begin
            gotwiad := deseq (kolgotowość),
            wywiad.treść := wewiad.treść;
            wywiad.odbiiorca := gotwiad.nadawca
            send (wywiad)
          end
        end
      else
        begin
          if empty (koldanych) then
            enseq (wewiad,kolgotowości)
          else
            begin
              wywiad:=deseq (koldanych);
              wywiad.odbiiorca := wewiad.nadawca
              send (wywiad)
            end
          end
        end
      end
    end
  end

```



```

        end
    end
endprocess;
process czytnik-i-ty;
var wiad : message informacja_uzytkowa;
        b : blok;
begin
    wiad.odbiorca := wspólny_bufor;
    while true do
        begin
            read (urządźwe_i-te, b);
            wiad.treść := b;
            send (wiad)
        end
    endprocess;
process pisak-k-ty;
var wiadster : message informacja_sterująca;
    dane : message informacja_uzytkowa;
begin
    wiadster.odbiorca := wspólny_bufor;
    while true do
        begin
            send (wiadster);
            dane := receive;
            write (urządźwy_k-te, dane.treść)
        end
    endprocess
endprogram.

```

Należy zwrócić uwagę, że wprowadzenie typu kolejkowego sequence, umożliwiającego tworzenie kolejek o nieograniczonej długości, nie ogranicza także liczby buforowanych bloków. Takie założenie może być w wielu wypadkach nierealne; bardziej szczegółowy przegląd innych sposobów komunikacji procesów przedstawiono w rozdziale 5.

Ogólne wnioski wypływające z analizy przygotowanych przykładów pozwalają stwierdzić następujące fakty. Siła ekspresji omawianego podejścia jest większa od modelu obliczeń w środowisku zwartym. Oznacza to możliwość transformacji każdego opisu w konwencji środowiska zwartego na opis w konwencji środowiska rozproszonego; nie zawsze natomiast jest możliwa transformacja odwrotna. Stopień strukturalizacji podejścia rozproszonego jest także większy niż w poprzednim: wynika to stąd, że

jawnie deklaruje się sposób komunikacji pomiędzy procesami. Jeżeli chodzi o względy implementacyjne, to nie można jednoznacznie określić, które z omawianych podejść jest wygodniejsze. Należy stwierdzić, że oba podejścia są wzajemnie się uzupełniające: tam, gdzie istotnie środowisko, w którym następuje implementacja, ma charakter zwarty, dogodniejsze jest podejście pierwsze, w przypadku środowiska rozproszonego natomiast podejście drugie.

2.5. Obliczenia równoległe synchronizowane przepływem danych

Omówione poprzednio podejścia programowania współbieżnego były oparte na koncepcji procesów sekwencyjnych komunikujących się przez wspólną pamięć lub przez przesyłanie wiadomości. Stąd wyrosłe języki programowania przypominają tradycyjne języki programowania sekwencyjnego z dodatkowym wyposażeniem w mechanizmy komunikacji i synchronizacji. Obliczenia równoległe synchronizowane przepływem danych stanowią zupełnie nowe podejście. Program jest tutaj traktowany jako pewien zbiór instrukcji, w którym tekstowa kolejność zapisu instrukcji nie wyznacza kolejności ich wykonywania. Zasada określająca kolejność wykonywania instrukcji - zasada sterowania - jest następująca: daną instrukcję można obliczać wówczas, gdy będą już obliczone argumenty tej instrukcji. Stąd wynika nazwa podejścia - obliczenia synchronizowane przepływem danych (synchronizowane gotowością argumentów; data flow computations), podczas gdy podejścia poprzednie można by określić jako obliczenia synchronizowane przepływem sterowania (control flow computations) [12,16,61].

Realizacja jednoznacznych obliczeń zgodnie z zasadą przepływu danych wymaga spełnienia dwóch warunków. Warunek pierwszy to brak efektów ubocznych wykonania instrukcji. Oznacza to, że jedynym efektem wykonania instrukcji musi być przypisanie pewnej wartości identyfikatorowi tej instrukcji, a nie mogą nastąpić zmiany przyporządkowania wartości innym identyfikatorom. Warunek drugi wymaga, aby przyporządkowanie wartości danemu identyfikatorowi mogło nastąpić tylko jednokrotnie.

Ze względu na fakt, że efekty uboczne, a zwłaszcza wielokrotne przypisywanie wartości zmiennej (identyfikatorowi instrukcji podstawienia), stanowią kamień węgielny klasycznego programowania, należy zdać sobie sprawę, że ich usunięcie z języka wymaga wprowadzenia w zamian mechanizmów. Są nimi: rekursja procedur oraz listy pośredniczące - krótko listy (streams). Rekursja eliminuje potrzebę iteracji - konstrukcji, która w zasadniczy sposób bazuje na efektach ubocznych (zmiana wartości wskaźnika) oraz na wielokrotnym wykorzystaniu instrukcji przypisania. Listy pośredniczące służą do grupowania w jeden wspólny obiekt ciągów danych elementarnych. Pozwalają one definiować jednostki programowe (analogowe

procedur), które wykonują operacje na ciągach danych. Tego typu jednostki, które mogą przetwarzać listy pośredniczące i/lub wytwarzać listy pośredniczące, będą nazywane modułami (modules). Na listach pośredniczących można wykonywać następujące operacje. Jeżeli lista a jest ciągiem:

$$s = x_1, x_2, \dots, x_n,$$

to:

$$\text{first}(s) = x_1,$$

$$\text{rest}(s) = x_2, x_3, \dots, x_n,$$

$$\text{cons}(y, s) = x_1, \dots, x_n, y.$$

Dla podkreślenia istotnej różnicy pomiędzy znaczeniem instrukcji przypisania (podstawienia) w stosunku do języków tradycyjnych wprowadza się oznaczenie:

$$\text{let} \langle \text{identyfikator} \rangle = \langle \text{wyrażenie} \rangle$$

zamiast

$$\langle \text{identyfikator} \rangle := \langle \text{wyrażenie} \rangle.$$

Ponadto jest potrzebny mechanizm przetwarzania tablic bez efektów ubocznych. Dlatego definiuje się funkcję modify (A, i, v), której wynikiem działania jest nowa tablica A' , różniąca się od tablicy A tylko tym, że na pozycji i -tej występuje wartość v .

P.1. Przykład systemu rezerwacji biletów, wbrew pierwszemu wrażeniu powodowanemu podobieństwami syntaktycznymi, ma zupełnie odmienną semantykę od języka Pascal. Początkowo skupimy uwagę na konstrukcji jednego modułu "transakcje", którego wejściami są: lista zleceń klientów

strumień_żądań : stream of żądanie

tablica lotów

miejsca dostępne : array [1..n] of integer,

wyjście zaś stanowi lista

strumień_odpowiedzi : stream of odpowiedź,

gdzie

type żądanie =

record

typ : (rezerwuj, informuj);

klient, lot, liczba : integer

end

oraz

type odpowiedź =

record

```

powodzenia : boolean;
klient, wolne : integer;
end

```

Moduł "transakcja" ma postać:

```

module transakcja (strumień_żądań:
  stream of żądanie; miejsca_dostępne:
  array [1..n] of integer);
returns strumień_odpowiedzi:
  stream of odpowiedź;
var wejście : żądanie;
    wyjście : odpowiedź;
    i,j : integer;
    pozostałe_miejsca : array [1..n] of integer;
begin
  let wejście = first (strumień_żądań);
  case wejście.typ of
    rezerwuj:
      begin
        let i = wejście.lot;
        let j = wejście.liczba;
        if miejsca_dostępne [i]  $\geq$  j then
          begin
            let wyjście.powodzenie = true;
            let pozostałe_miejsca =
              modify (miejsca_dostępne, i,
                miejsca_dostępne [i] - j)
          and
          else
            begin
              let wyjście.powodzenie = false;
              let pozostałe_miejsca =
                miejsca_dostępne
            end
          end;
        informuj:
          begin
            let i = wejście.lot;
            let pozostałe_miejsca = miejsca_dostępne
          end
          let wyjście.klient = wejście.klient;
          let strumień_odpowiedzi =

```



```

cons (wyjście, transakcja (rest
    (strumień żądań), pozostałe_miejsca))
endmodule.

```

Moduł "transakcja" otrzymuje dane wejściowe w postaci jednego strumienia żądań, którego elementy mogą pochodzić z różnych miejsc (końcówek). Na razie nie omówiono sposobów tworzenia takiej listy. Przedstawiony język służył do wyrażania tylko zdeterminowanych obliczeń, podczas gdy system rezerwacji zachowuje się w sposób niezdeterminowany. Dla wyrażenia niedeterminizmu wprowadza się operację scalania list pośredniczących merge. Działając na listach s_1 , s_2 operacja merge (s_1, s_2) tworzy nową listę pośredniczącą, złożoną ze wszystkich elementów obu list, w taki sposób, że zostaje zachowana względna kolejność elementów z s_1 oraz s_2 , natomiast sposób przemieszczania się elementów z s_1 względem elementów z s_2 jest niezdeterminowany. Operacja merge jest wystarczająca do przedstawienia szerokiej klasy obliczeń niedeterministycznych. Przykładowo przy założeniu, że system rezerwacji ma trzy końcówki (trzy moduły), które produkują listy: $lista_żąd_1$, $lista_żąd_2$, $lista_żąd_3$, moduł, który obsługuje te listy, będzie miał postać:

```

module system ( $lista\_żąd\_1$ ,  $lista\_żąd\_2$ ,
     $lista\_żąd\_3$ : stream of żądanie;
     $miejsca\_dostępne$ : array [1..n] of integer)
returns  $lista\_odp\_1$ ,  $lista\_odp\_2$ ,  $lista\_odp\_3$ :
    stream of odpowiedź;
    var  $p_1, p_2, p_3$ : stream of żądanie;
    begin
        let  $p_1$  = tag ( $lista\_żąd\_1, 1$ );
        let  $p_2$  = tag ( $lista\_żąd\_2, 2$ );
        let  $p_3$  = tag ( $lista\_żąd\_3, 3$ );
        let  $lista\_żądań$  = merge ( $p_1$ , merge ( $p_2, p_3$ ));
        let  $lista\_odpowiedzi$  =
            transakcja ( $lista\_żądań$ ,  $miejsca\_dostępne$ );
        let  $lista\_odp\_1$ ,  $lista\_odp\_2$ ,  $lista\_odp\_3$  =
            sort (strumień\_odpowiedzi)
    endmodule.

```

W module "system" elementy wejściowych list pośredniczących są etykietowane przez operację tag. tzn. każdemu elementowi takiej listy są przyporządkowane odpowiednio liczby 1,2,3. Etykiety stanowią rozróżnienie elementów pochodzących z różnych źródeł, dzięki czemu po otrzymaniu listy "strumień_odpowiedzi" z modułu "transakcja" można wydzielić przez operację sort, w wypadku omawianego podejścia, podlisty kierowane do odpowiednich końcówek.

P.2. Przykład systemu buforowania nie wnosi nowych elementów, dlatego pozostawia się go jako ćwiczenie dla Czytelnika.

Języki programowania sterowania przepływem danych znajdują się aktualnie w początkowej fazie rozwoju. Wydają się one bardzo obiecujące dla obliczeń równoległych ze względu na dużą siłę ekspresji oraz wysoki stopień modularności programowania. W pewnym sensie można je uznać za krok w kierunku języka klasycznej matematyki: chodzi tu o to, że język matematyki nie operuje pojęciem zmiennej rozumianej jako komórka pamięci, której zawartość można modyfikować, natomiast rekursja (indukcja) jest dobrze znanym mechanizmem. Podział programu na moduły, które mogą oddziaływać na siebie w jawnie zdefiniowany sposób, jawne wskazanie źródeł niedeterminizmu (operator merge) czynią program czytelnym, pomimo swej "inności" w stosunku do powszechnie stosowanych podejść.

Implementacja języków synchronizowanych przepływem danych jest raczej na prymitywnym etapie. Wynika to przede wszystkim z wymagań całkowicie odmiennej architektury sprzętu liczącego. Poszukiwania odpowiedniej architektury oraz badania tych języków są ciągłym przedmiotem studiów [2,16,17,18,43,45,61].

2.6. Podsumowanie

W rozdziale przedstawiono trzy podstawowe koncepcje do programowania równoległego, w istotny sposób różniące się swymi podstawami. Pierwsza oparta na modelu zwartego środowiska traktuje program jako zbiór procesów działających pod wspólnym "dachem" i komunikujących się przez oddziaływanie na wspólne otoczenie. Druga ze środowiskiem rozproszonym traktuje program jako zbiór procesów, z których każdy działa pod własnym "dachem", komunikujących się przez przesyłanie wiadomości. W trzeciej koncepcji z przepływem danych program jest widziany jako sieć operatorów, z których każdy produkuje nową wartość po otrzymaniu kompletu wartości wejściowych argumentów. Sieć ta może rekurencyjnie rozwijać się i związać w zależności od wywoływania i zakończenia działania procedur - modułów.

Omówione koncepcje różnią się stopniem równoległości, jaki można osiągnąć, czytelnością i problemami implementacyjnymi.

Żaden program złożony z komunikujących się procesów sekwencyjnych nie realizuje pełnego, możliwego do osiągnięcia poziomu równoległości. W programach z semaforami liczba procesów równoległych zależy w istocie od pomysłowości programisty; w programach z monitorami należy dokonać kompromisu pomiędzy całkowitą ochroną zasobów (a przez to wyłączenie jednoczesnego dostępu do zasobów) a dopuszczaniem procesów do bezpośred-

niego dostępu. W procesach komunikujących się zasoby są bezpośrednio dostępne tylko jednemu procesowi; dlatego, jeżeli zasoby są podzielone pomiędzy różne procesy, równoległość jest ograniczona. W językach synchronizowanych przepływem danych można wyrazić wszelkie formy równoległości dopuszczalne przez algorytmy.

Ocena czytelności programów jest zagadnieniem subiektywnym, jednakże takie własności, jak modularność, ograniczenie źródeł niedeterminizmu, konstrukcje językowe wysokiego poziomu są lepiej uchwytne. Kolejność omawianych podejść odpowiada wzrastającemu stopniowi modularności. Semaforey, jako narzędzie bardziej elementarne, nie wprowadzały bezpośrednio żadnej modularności, pozwalając procesom na interakcję łatwo wymykającą się kontroli programisty. Wyższy stopień prezentują programy z procesami komunikującymi się. Najwyższy stopień osiągają programy synchronizowane przepływem danych, w których dzięki całkowitej eliminacji efektów ubocznych działanie każdego modułu definiuje się jako przetworzenie pośredniczących list wejściowych na wyjściowe; podejście to jako jedyne wprowadza też zasadę jawnego zaznaczania źródeł niedeterminizmu.

Ze względu na problemy implementacyjne, kolejność prezentacji podejść odpowiada rosnącej skali problemów. Stwierdzenie to obrazuje fakt, że semaforey i monitory (oraz inne tej klasy konstrukcje) znalazły implementację w różnych językach programowania, w mniejszym stopniu dotyczy to procesów komunikujących się, podczas gdy implementacja języków synchronizowanych przepływem danych znajduje się w fazie rozwiązań koncepcyjnych.

ĆWICZENIA

1. Zadanie o czytelnikach i pisarzach (Courtois, Heymens, Parnas). Pewna liczba procesów pisarzy prowadzi zapis informacji do bufora, z którego informacja jest odczytywana przez procesy czytelników. Do buforu może mieć w danej chwili dostęp dowolna liczba czytelników, ale tylko jeden pisarz. Ponadto pisarze mają priorytet przed czytelnikami. Przenalizuj poprawność przedstawionego niżej rozwiązania z wykorzystaniem semaforów.

```
integer lczyt, lpis,  
semaphore m1,m2,m3,p,c;  
lczyt:=0; lpis:=0;  
m1:=1; m2:=1; m3:=1; p:=1; c:=1;
```

```
process czytelnik;  
begin  
wait (c);
```

```
process pisarz;  
begin  
wait (m2);
```

```

wait (m1);
lczyt:=lczyt+1;
if lczyt = 1 then
    wait (p);
signal(c);
signal(m3);
czytaj z bufora;
wait(m2)
lczyt:=lczyt-1;
if lczyt = 0 then
signal(p);
signal(m1);
end

lpis:=lpis+1;
if lpis = 1 then
    wait (c);
signal(m2);
wait (p);
czytaj z bufora;
signal(p);
wait(m2);
lpis:=lpis-1;
if lpis = 0 then
signal(c);
signal(m2);
end

```

2. Rozwiąż zadanie o czytelnikach i pisarzach (ćw. 1) z wykorzystaniem monitorów.

3. Rozwiąż zadanie o czytelnikach i pisarzach z wykorzystaniem procesów komunikujących się przez przesyłanie wiadomości kanałami komunikacyjnymi.

4. Przeanalizuj działanie podanego niżej monitora, stanowiącego odpowiednik monitora "wspólny bufor", przedstawionego w p. 2.3.

```

monitor bufor_we_wy;
var bufor : blok;
używany : boolean;
wolny, zajęty : queue;
procedure entry wpis (do : blok);
begin
    if używany then delay (wolny);
    bufor := do;
    używany := true;
    continue (zajęty)
end;
procedure entry odczyt (var z : blok);
begin
    if not używany then delay (zajęty);
    z := bufor;
    używany := false;
    continue (wolny);
end;
begin używany := false end.

```

Wykaż, że procesy wywołujące procedury wejściowe tego monitora mogą być zawieszane bez możliwości odwieszenia.

5. Zadanie o filozofach (Dijkstra). Dookoła okrągłego stołu siedzi pięciu filozofów. Na stole znajduje się pięć widelców, każdy pomiędzy dwoma sąsiadującymi filozofami. Filozofowie znajdują się na przemian w jednej z faz: "myślenie", które sprowadza się tutaj do bezczynnego oczekiwania oraz "jedzenie", które polega tu na uchwyceniu widelca z lewej i z prawej strony do jedzenia spaghetti znajdującego się na talerzu. Oczywiście nie zawsze potrzebne widelce są wolne i w takiej sytuacji filozof musi czekać na ich zwolnienie.

Zaproponuj zasady wspólnego korzystania z widelców przez filozofów w taki sposób, aby żaden z nich nie musiał oczekiwać na jedzenie nieskończenie długo. Przedstaw to rozwiązanie w postaci programu z użyciem semaforów.

6. Rozwiąż zadanie o filozofach (ćw. 5) z wykorzystaniem monitorów.

7. Warunkowe regiony krytyczne (Brinch Hansen) stanowią jedną z propozycji mechanizmu synchronizacji dostępu do wspólnych zasobów. Konstrukcja

region v do S end

związuje ze zmienną (obiektom) v anonimowy semafor, wykluczający dostęp do regionu krytycznego S, który stanowi jedyne miejsce dostępu do v. Warunek await B wewnątrz regionu zawieszca czasowo wołający program aż do momentu, gdy B stanie się prawdziwe. Dopóki B jest fałszywe proces zostaje zawieszony, a region krytyczny staje się otwarty dla innych procesów. Proces zostaje wznowiony, gdy B stanie się prawdziwe, a region zostaje ponownie zamknięty dla innych procesów. Przeanalizuj wady i zalety tego rozwiązania z punktu widzenia czytelności programów i implementacji.

8. Rozwiąż zadania o czytelnikach i pisarzach (ćw. 1) oraz o filozofach (ćw. 5) z wykorzystaniem warunkowych regionów krytycznych.

9. Wyrażenia ścieżkowe (Campbell, Habermann) stanowią mechanizm synchronizacji, którego istota polega na określeniu dopuszczalnych klas oddziaływań (historii wykonania) współdziałających procesów. Wyrażenie ścieżkowe ma postać

path S end

gdzie S może oznaczać różne dopuszczalne kolejności wykonania:

$S ::= S1; S2$

oznacza, że S1 musi poprzedzać wykonanie S2 oraz odwrotnie;

$S ::= S1, S2$

oznacza, że może być wykonana albo S1 albo S2;

$S ::= \{S1\}$

oznacza, że po zainicjowaniu S1 może być zainicjowana dowolna liczba egzemplarzy S1; dalsze obliczenia można kontynuować dopiero po zakończeniu wszystkich tych egzemplarzy

$S ::= (a-b)^n$

oznacza, że w dowolnym momencie wykonywania programu sekwencja "a" musi być wykonana co najmniej tak często, jak sekwencja "b" oraz nie więcej niż n razy częściej. Zaproponuj systematyczną metodę zamiany wyrażeń ścieżkowych na odpowiednie działania na semaforach.

10. Rozwiąż zadania o czytelnikach i pisarzach (ćw. 1) oraz o filozofach (ćw. 5), wykorzystując wyrażenia ścieżkowe.

3. JĘZYK PROGRAMOWANIA CONCURRENT PASCAL

3.1. Wprowadzenie

Język Concurrent Pascal należy do klasy języków bazujących na założeniu wspólnego środowiska dla równoległe pracujących procesów. Prace nad językiem prowadził P. Brinch Hansen na Uniwersytecie Kalifornijskim w latach 1972-75. (Pierwsze koncepcje języka były już dyskutowane w jego książce "Podstawy systemów operacyjnych" [8].) Pierwszy kompilator języka na minikomputer PDP 11/45 opracował Hartman w 1975 r. [27]. Concurrent Pascal jest pierwszym w swej klasie językiem wysokiego poziomu i, obok Moduli, aktualnie najbardziej rozpowszechnionym. Do upowszechnienia przyczynił się m.in. fakt, że kompilator języka został napisany w języku maszyny wirtualnej, co umożliwia stosunkowo proste przeniesienie oprogramowania na inne komputery. (Wymaga to napisania programu bazowego - jądra - o objętości rzędu 3-4 K słów, modelującego działanie maszyny wirtualnej.)

Nazwa języka oddaje częściowo jego charakter. Concurrent Pascal jest oparty na uproszczonej wersji języka Pascal, wyposażonej w dodatkowe konstrukcje umożliwiające definiowanie procesów i sposobów ich współpracy. Podstawowe konstrukcje, które dołączono, to proces, monitor i klasa. Konstrukcje procesu i monitora są podobne do analogicznych konstrukcji omówionych w rozdz. 2, konstrukcja klasy stanowi natomiast uproszczoną wersję klasy z języka Simula i stanowi mechanizm modularyzacji programów.

Ogólnie program w Concurrent Pascalu przedstawia się w postaci hierarchicznie uporządkowanego zbioru komponentów. Komponentami (zmiennymi systemowymi) są właśnie procesy, monitory oraz klasy. Ważną cechą języka, wpływającą na podniesienie poziomu strukturalizacji programów, jest wprowadzenie pojęcia typów systemowych (typy procesowe, monitorowe i klasowe). Daje to możliwość definiowania wielu egzemplarzy zmiennych systemowych (procesów, monitorów, klas) tego samego typu. Przewidziany jest także mechanizm parametryzacji komponentów w obrębie jednego typu; jest on realizowany przez tzw. instrukcje inicjalizacji, które dokonują m.in. ustalenia wartości parametrów każdego komponentu programu, a w wyniku tego zostaje ustalone uporządkowanie pomiędzy komponentami. Pojęcie porządku pomiędzy komponentami odnosi się do zasad komunikowania się (zasad dostępu) komponentów.

3.2. Symbole podstawowe

Elementami alfabetu języka są znaki należące do kodu ASCII (American Standard Code of Information Interchange), symbole specjalne, identyfikatory, komentarze i słowa kluczowe. A zatem są to:

a) znaki graficzne (te, które mają reprezentację graficzną) kodu ASCII:

- litery A B C ... Z

- cyfry 0 1 2 ... 9

- znaki specjalne

! " # \$ % & ' () * + , - . / : ; < = > ? @

b) znaki sterujące kodu ASCII; znaki te nie mają reprezentacji graficznej, a w programie są przedstawiane w postaci liczby z zakresu 0..127, będącej kodem, umieszczonej pomiędzy znakami (: oraz :), np. (: 10 :) jest reprezentacją znaku sterującego zmianą wiersza,

c) symbole specjalne, kombinacje par znaków z kodu ASCII:

< > oznaczający ≠

< = oznaczający ≤

> = oznaczający ≥

(. oznaczający [

.) oznaczający]

:= oznaczający symbol podstawienia

.. oznaczający separator wskaźników,

d) identyfikatory, stanowiące - jak zwykle - ciąg liter lub cyfr zaczynający się literą;

e) separatory: spacja (: 32 :), nowa linia (: 10 :), komentarz - dowolny ciąg znaków nie zawierający " zamknięty w cudzysłowy;

f) słowa kluczowe:

<u>array</u>	<u>begin</u>	<u>case</u>	<u>class</u>	<u>const</u>
<u>cycle</u>	<u>div</u>	<u>do</u>	<u>downto</u>	<u>else</u>
<u>end</u>	<u>entry</u>	<u>for</u>	<u>function</u>	<u>if</u>
<u>in</u>	<u>init</u>	<u>mod</u>	<u>monitor</u>	<u>not</u>
<u>of</u>	<u>or</u>	<u>procedure</u>	<u>process</u>	<u>program</u>
<u>record</u>	<u>repeat</u>	<u>set</u>	<u>then</u>	<u>to</u>
<u>type</u>	<u>univ</u>	<u>until</u>	<u>var</u>	<u>while</u>
<u>with</u>				

3.3. Struktury danych

Pascalowe typy danych oraz związane z nimi operatory występujące w Concurrent Pascalu są następujące:

- typ wyliczeniowy z operacjami (funkcje standardowe) poprzednika (przed) oraz następnika (succ);
- typ całkowity (integer) z operacjami : + , - , * , div, mod oraz funkcjami standardowymi wartość bezwzględna (abs), zamiana liczby kodu na znak (chr), zamiana liczby całkowitej na rzeczywistą (conv);
- typ rzeczywisty (real) z operacjami: + , - , * , / oraz funkcjami standardowymi: wartość bezwzględna (abs), zaokrąglenie liczby rzeczywistej do liczby całkowitej przez odrzucenie części ułamkowej (trunc);
- typ logiczny (boolean) z operacjami: & , or, not;
- typ znakowy (char) z operacją zamiany znaku na liczbę całkowitą -
- kod znaku (ord);

(Do wszystkich wymienionych wyżej typów prostych odnoszą się, w znany sposób, operatory relacji =, <>, <=, >=, <, > oraz operator podstawienia := . Pojęcie typu okrojonego jest takie same, jak w Pascalu.)

- typ tablicowy (array) i typ rekordowy (record) z operatorami: =~~≠~~;

(Uwaga: typ rekordowy nie zawiera wariantów);

- typ zbiorowy (set) z operacjami: & (przekrój), or (suma), in (przynależność do zbioru) oraz =, <>, >=, <=, <, > , := .

Concurrent Pascal nie zawiera pascalowych typów pliku i typów wskaźnikowych.

Niepascalowe typy danych: procesy (process), monitory (monitor), klasy (class) i inne dodatkowe typy są zdefiniowane w dalszej części rozdziału - punkty 3.5-3.7.

Deklaracja typów (type), stałych (const) oraz zmiennych (var) w odniesieniu do omówionych typów pascalowych jest taka sama, jak w Pascalu.

3.4. Instrukcje

Zbiór instrukcji zawiera następujące instrukcje, znane z języka Pascal:

- przypisanie (podstawienie),
- pusta,
- złożona (begin J₁; J₂:...; J_n end);
- warunkowa (if);
- wyboru (case);
- iteracji (for, while oraz repeat),
- wiążąca (with).

Dodatkowo wprowadzono instrukcję nieskończonej iteracji

cycle J end

równoważnej pascalowej konstrukcji

while true do J.

Oddzielnego omówienia wymaga instrukcja procedury. Pojęcie procedury jest używane w szerszym kontekście niż w Pascalu. Ogólnie procedury występują w dwóch kontekstach: jako jednostki strukturalizacji wewnętrznej programów (tak jak w Pascalu) oraz jako atrybuty komponent (tzn. procesów, monitorów i klas) programu równoległego. W tym pierwszym sensie procedury dzieli się na:

- procedury właściwe (procedure),
- funkcje (function),
- programy sekwencyjne (program).

Wprowadzenie programów sekwencyjnych jako pewnego rodzaju procedury ma głębsze uzasadnienie. Jednym z głównych zastosowań Concurrent Pascala jest programowanie systemów operacyjnych. Działanie systemu operacyjnego polega m.in. na zarządzaniu sekwencyjnymi programami użytkowników. Dlatego potrzeba takich mechanizmów, które pozwalałyby na pobieranie tych programów z pamięci pomocniczej do pamięci operacyjnej (i odwrotnie) i ich wykonywanie. Z tymi zadaniami wiąże się koncepcja programu sekwencyjnego, jako szczególnego rodzaju procedury, wywołanie takich procedur odpowiada rozpoczęciu obliczeń takiego programu pod nadzorem komponentu (tu może to być tylko proces), który go wywołał.

Podział procedur w drugim sensie, jako atrybuty komponentów programu, wydziela następujące rodzaje procedur:

- procedury wejściowe procesów,
- procedury wyjściowe monitorów,
- procedury wejściowe klas,
- instrukcje inicjalizacji procesów, monitorów i klas.

Sens i sposób użycia tej grupy procedur opisano w punktach 3.5-3.7.

Komunikacja pomiędzy procedurami a wywołującymi je komponentami odbywa się:

- w przypadku procedur wejściowych oraz inicjalizacji wyłącznie poprzez parametry,
- w przypadkach pozostałych dodatkowo jeszcze za pomocą zmiennych nielokalnych w treści procedury.

Komunikacja poprzez parametry odbywa się poprzez zmienne (odpowiedni parametr w definicji procedury jest poprzedzony słowem var) lub przez wartość - tak jak w Pascalu. Specyficzną cechą tej komunikacji jest możliwość wykorzystania tzw. typu uniwersalnego (univ). Jego wprowadzenie wiąże się z tą samą potrzebą, dla której wprowadzono programy sekwencyjne jako szczególny rodzaj procedury. Wywołanie procedury $P(x)$, której nagłówek został zadeklarowany w postaci:

$P(v: \text{univ } T)$

lub

$P(\text{var } v : \text{univ } T)$

pociąga za sobą następujący skutek. Jeżeli parametr aktualny x jest wartością lub zmienną typu $T1$, to uważa się, że typy T oraz $T1$ są zgodne (i wywołanie jest prawidłowe), jeżeli kody obu typów są reprezentowane przez tę samą liczbę komórek pamięci. Typ T może być dowolnym typem passywnym (nie może być procesem, monitorem lub klasą). Opisany mechanizm stwarza np. możliwość przesyłania różnych obszarów pamięci pomocniczej do pamięci operacyjnej i reprezentowania ich za pomocą jednego typu: w szczególności różne programy sekwencyjne sprowadzone do pamięci operacyjnej można zaprezentować za pomocą jednego typu.

W stosunku do Pascala wszystkie procedury obowiązują ograniczenie: procedury nie mogą wywoływać się rekursywnie oraz definicje procedur nie mogą być w sobie zagnieżdżane. Ograniczenie to nie dotyczy programów sekwencyjnych (program) napisanych w Pascalu, a wykonywanych pod nadzorem procesów napisanych w Concurrent Pascalu.

3.5. Procesy

Proces - oprócz monitora i klasy - stanowi jeden z obiektów systemowych. Zasadniczo definicja typu procesowego (bardziej rozbudowaną definicję procesu zdefiniowano przy końcu bieżącego punktu) przypomina definicję programu w Pascalu i ma postać

```
type <identyfikator typu procesowego > =
    process (<wykaz parametrów formalnych>;
    <blok>
```

Podobnie wygląda deklaracja zmiennych typu procesowego (zmiennych systemowych):

```
var<identyfikator zmiennej> :
    <identyfikator typu procesowego>
```

Przez <blok> rozumie się, dokładnie tak jak w Pascalu, strukturę postaci:

```
<deklaracja obiektów lokalnych>;
<instrukcja złożona>
```

gdzie <deklaracja obiektów lokalnych> jest ciągiem:

```
<deklaracja stałych>;
<deklaracja typów>;
<deklaracja zmiennych>;
<deklaracja procedur i funkcji>
```

Proces jako typ systemowy definiuje abstrakcyjną strukturę danych złożoną z:

- lokalnej (prywatnej) struktury danych, którą stanowią zadeklarowane stałe typy, zmienne, funkcje i procedury; obiekty te nie są dostępne dla komponentów (zmiennych systemowych) zdefiniowanych na zewnątrz danego procesu;

- pewnego programu sekwencyjnego (<instrukcja złożona>) działającego na tej strukturze.

Ponadto przez <wykaz parametrów formalnych> definiuje on zasady dostępu (zasady komunikacji) do pozostałych zmiennych systemowych (bezpośrednio do pewnych monitorów lub klas, a pośrednio do innych procesów).

Dostęp procesu do pewnych monitorów lub klas oznacza, że proces może wywoływać procedury wejściowe tych monitorów lub klas (p. 3.6 i 3.7). Przykładowo, jeżeli

```
type P = process (m : M)
          <blok>,
```

gdzie M jest pewnym typem monitorowym, to oznacza, że procesy typu P mogą wewnątrz <bloku> wywoływać procedury wejściowe pewnego monitora typu M.

Deklaracja procesu jako zmiennej (systemowej) stanowi tylko rezerwację obszaru pamięci wymaganego dla zapamiętania jego struktury danych i ciągu jego instrukcji. Natomiast ostatecznym uformowaniem procesu - ustaleniem konkretnych wartości dla jego parametrów formalnych i wystartowaniem obliczeń zajmuje się instrukcja inicjalizacji postaci:

```
init <identyfikator zmiennej procesowej>
      (<wykaz parametrów aktualnych>).
```

Przykładowo, jeżeli mamy zmienną procesową p, krótko proces, zadeklarowaną przez

```
var p : P,
```

to instrukcja inicjalizacji

```
init p (a)
```

spowoduje wystartowanie obliczeń procesu p, podczas których proces p może odwoływać się do monitora a typu M.

Instrukcję inicjalizacji można porównać do wywołania procedury. Podobieństwo to jest pełne w zakresie przekazywania informacji przez parametry. Parametry mogą być komunikowane:

- przez wartość w odniesieniu do parametrów typu passywnego,
- przez nazwę w odniesieniu do parametrów typu systemowego (tak jak w ostatnim przykładzie).

Brak natomiast tego podobieństwa do wywołania procedury, w zakresie przekazywania sterowania. Wywołanie procedury powoduje przekazanie jej sterowania, a zakończenie obliczeń zwrot sterowania do miejsca, w którym nastąpiło jej wywołanie. Inicjalizacja procesu rozpoczyna obliczenia, trwające niezależnie (równolegle) od tych obliczeń, w których nastąpiła inicjalizacja procesu.

Gdy proces spełnia rolę nadzorującą w stosunku do pewnych programów sekwencyjnych (program), może zawierać on procedury lub funkcje wejściowe, które mogą być wywołane przez te programy. Składnia typu procesu może więc mieć bardziej rozbudowaną postać:

```
type <identyfikator typu procesowego> =
  process (<wykaz parametrów formalnych>;
    <deklaracje obiektów lokalnych>;
    <deklaracje procedur wejściowych>;
    <instrukcja złożona>;
```

gdzie <deklaracja procedur wejściowych> jest taka, jak w wypadku procedur wejściowych monitora (p. 3.6).

- 3.6. Monitory

Monitor stanowi obiekt systemowy. Definicja typu monitorowego ma postać:

```
type <identyfikator typu monitorowego> =
  monitor (<wykaz parametrów formalnych>;
    <deklaracje obiektów lokalnych>;
    <deklaracja procedury wejściowej>;
    .....
    <deklaracja procedury wejściowej>;
    <instrukcja inicjalizująca>.
```

Znaczenie poszczególnych elementów podanej definicji jest następujące: <deklaracja obiektów lokalnych> jest rozumiana tak, jak w poprzednim p. 3.5; <deklaracja procedury wejściowej> ma postać:

```
procedure entry <identyfikator>
  (<wykaz parametrów formalnych>;
    <blok>;
```

czyli od zwykłej procedury różni się syntaktycznie tylko dołączeniem słowa entry po słowie procedura lub function.

Monitor, jako typ systemowy, definiuje abstrakcyjną strukturę danych złożoną z:

- lokalnej struktury danych, które są bezpośrednio dostępne wyłącznie nie procedurom wejściowym i instrukcji inicjalizującej monitora, a pośrednio - dzięki dopuszczeniu wywoływania procedur wejściowych przez inne komponenty programu - równolegle pracującym procesom;

- dowolnej liczby procedur wejściowych, które wywołane tylko przez inne komponenty działają na wspólnej, lokalnej strukturze danych;

- instrukcji inicjalizującej (instrukcja złożona) służącej do ustalania początkowych wartości lokalnej struktury danych podczas inicjalizacji monitora.

Ponadto, poprzez <wykaz parametrów formalnych>, definiuje on zasady dostępu do innych zmiennych systemowych, rozumiane tak samo, jak sformułowano to w punkcie 3.5.

Definicja zmiennej typu monitorowego ma postać:

```
var <identyfikator zmiennej monitorowej>:  
    <identyfikator typu monitorowego>
```

i znaczenie podobne przy deklaracji procesu: jest rezerwacją odpowiedniego obszaru pamięci.

Ostateczne uformowanie monitora polega na wykonaniu instrukcji inicjalizacji:

```
init <identyfikator zmiennej monitorowej>  
    (<wykaz parametrów aktualnych>),
```

która powoduje zastąpienie parametrów formalnych przez aktualne (jak dla procesu) oraz wykonanie złożonej instrukcji inicjalizującej (beziemenna procedura) ustalającej początkowe wartości lokalnej struktury danych. Dopiero po wykonaniu inicjalizacji monitor staje się dostępny innym komponentom.

Monitor jest komponentem biernym (usługowym, tzn. samodzielnie nie inicjuje żadnych obliczeń, a jedynie wykonuje obsługę kierowanych do niego żądań (wywołań procedur wejściowych). Wywoływania procedur wejściowych mają postać:

```
<identyfikator monitora>.<identyfikator procedury  
wejściowej>(<wykaz parametrów aktualnych>)
```

Monitor stanowi mechanizm synchronizacji dostępu procesów (ogólnie różnych komponentów programu) do wspólnych danych - struktury lokalnej monitora. Swe funkcje synchronizujące monitor realizuje przez dwa mechanizmy (opisane wstępnie w p. 2.3.2).

Pierwszy - nazywany mechanizmem szeregowania krótkoterminowego - jest wbudowany w definicję monitora. Rozstrzyga on o sposobie szeregowania napływających wywołań procedur wejściowych. Jeżeli monitor jest wolny (nie wykonuje żadnej procedury wejściowej), to pierwsze zgłoszenie,

które napłynęło do monitora zajmuje go na okres realizacji wybranej procedury wejściowej. Jeżeli w czasie, gdy monitor jest wolny napłynęło kilka zgłoszeń, to tylko jedno z nich zostaje wybrane do realizacji, a pozostałe oczekują na ponowne zwolnienie monitora. Zwolniony monitor wybiera jedno (wybór niedeterministyczny) z oczekujących zgłoszeń i kieruje je do wykonania. Zgłoszenia, które napływają do monitora w czasie jego zajętości są włączane do zbioru zgłoszeń oczekujących na realizację. Ogólnie, mechanizm ten realizuje zasadę, że w określonym czasie tylko jeden proces ma dostęp do danych chronionych przez monitor.

Drugi - nazywany mechanizmem szeregowania średnioterminowego - może być programowany przez wykorzystanie zmiennych typu kolejkowego (queue). Ogólny sens tego mechanizmu polega na tym, że procesy w trakcie wykonywania wywołanych przez nie procedur wejściowych monitora mogą być zawieszane (zawieszanie jest rejestrowane przez zapamiętanie nazwy takiego procesu w zmiennej kolejkowej). Zawieszenie może trwać przez pewien odcinek czasu aż do momentu, gdy inny proces, wywołując monitor, dokona odwołania procesu. Sens zawieszenia się procesu polega na tym, że dany proces nie może kontynuować obliczeń ponieważ nie jest spełniony oczekiwany przez niego warunek, który może być spełniony tylko w wyniku działania innych procesów. Interpretacja takich warunków zależy oczywiście od konkretnego zastosowania.

Typ kolejkowy (queue) jest typem standardowym, a zmienne tego typu mogą być deklarowane wyłącznie wewnątrz monitora. Na zmiennych kolejkowych dopuszcza się trzy operacje zdefiniowane przez procedury standardowe: empty, delay, continue. Jeżeli x jest zmienną typu queue, to

- empty (x) jest funkcją boolowską, która przyjmuje wartość true, gdy zmienna x jest pusta (nie przechowuje identyfikatora żadnego procesu);

- delay (x) jest procedurą, której wywołanie (w czasie wykonywania pewnej procedury wejściowej monitora) powoduje: (a) wpisanie do x identyfikatora procesu mającego aktualnie dostęp do monitora, (b) zawieszenie tego procesu, (c) zwolnienie monitora do obsługi dalszych zgłoszeń;

- continue (x) jest procedurą, której wykonanie powoduje, że (a) proces, który miał aktualnie dostęp do monitora zwalnia monitor (zostaje w tym momencie zakończone wykonywanie wywołanej przez niego procedury wejściowej), (b) jeżeli kolejka x nie jest pusta, to proces pamiętany w x zostaje odwołany i uzyskuje dostęp do monitora (zostaje wznowiona od miejsca ostatniego zawieszenia tego procesu w procedurze wejściowej); jeżeli kolejka x jest pusta, to monitor staje się wolny dla innych zgłoszeń.

3.7. Klasy

Klasa - ostatni obiekt systemowy - syntaktycznie różni się bardzo nieznacznie od monitora. Definicja typu klasowego przedstawia się następująco:

```

type <identyfikator typu klasowego> =
  class (<wykaz parametrów formalnych>;
    <deklaracja obiektów lokalnych>;
    <deklaracja procedury wejściowej>;
    .....
    <deklaracja procedury wejściowej>;
    <instrukcja inicjalizująca>,

```

a zatem różni się zastąpieniem słowa monitor słowem class. Jeszcze jedną różnicą syntaktyczną jest możliwość deklaracji obiektów lokalnych, tzw. zmiennych wejściowych. Są one deklarowane, po zwykłych zmiennych, w postaci:

```

var entry <identyfikator zmiennej>:<typ>,

```

czyli od deklaracji zwykłych zmiennych różnią się dołączeniem słowa entry.

Klasa, jako typ systemowy, definiuje abstrakcyjną strukturę danych złożoną z:

- lokalnej struktury danych, które - podobnie jak w monitorze - są bezpośrednio dostępne tylko swoim procedurom wejściowym i swej instrukcji inicjalizującej, a pośrednio przez możliwość wywoływania procedur wejściowych na zewnątrz klasy;
- dowolnej liczby procedur wejściowych, które mogą być na zewnątrz wywoływane tylko przez jeden komponent,
- instrukcji inicjalizującej (instrukcja złożona), służącej do ustalania początkowych wartości lokalnej struktury danych.

Ponadto przez <wykaz parametrów formalnych> definiuje się zasady dostępu do innych zmiennych systemowych, rozumiane to jest tak, jak dla poprzednio omówionych komponentów.

Definicja zmiennej typu klasowego, jej inicjalizacja oraz postać wywołań procedur wejściowych wyglądają dokładnie tak, jak w przypadku monitora.

Zasadniczą różnicą dzielącą klasę od monitora jest to, że klasa nie stanowi mechanizmu synchronizacji dostępu procesów do wspólnych danych. Wynika to stąd, że klasa jest przeznaczona do ochrony danych dostępnych tylko jednemu komponentowi (temu komponentowi, wewnątrz którego jest zdefiniowana). Klasa stanowi tutaj uogólnienie zwykłej procedury.

Zmienne wejściowe (var entry), które można definiować wewnątrz klasy, można wywoływać (podobnie jak procedury wejściowe) na zewnątrz klasy, w postaci:

<identyfikator klasy>.<identyfikator zmiennej wejściowej>

Wywołanie takie może wyłącznie służyć odczytaniu wartości tej zmiennej, a nie może zmienić jej wartości. Pojęcie zmiennych wejściowych - chociaż może być zastąpione odpowiednią procedurą wejściową, która tylko odczytuje wartość wskazanej zmiennej - wprowadzono ze względu na często spotykaną w zastosowaniach potrzebę odczytania zmiennych bez naruszania ich wartości.

3.8. Struktura programu

Program, z punktu widzenia syntaktyki, jest blokiem zakończonym kropką (<blok>.). Blok ten będzie nazywany blokiem początkowym i należy traktować go jako wyróżniony, bezimienny proces, wewnątrz którego są zdefiniowane podstawowe komponenty programu i jest dokonana ich inicjalizacja. Wykonanie tego bloku nie wymaga inicjalizacji. Podstawowymi komponentami, które muszą być zdefiniowane w bloku początkowym są wszystkie procesy i wszystkie monitory programu.

Oznacza to, że wewnątrz zmiennych procesowych i monitorowych nie mogą być zdefiniowane inne zmienne procesowe lub monitorowe. Zmienne klasowe mogą natomiast być definiowane wewnątrz każdego komponentu; w szczególności oznacza to, że mogą one być zagnieżdżane.

Blok początkowy programu definiuje więc zbiór podstawowych komponentów, a ponadto definiuje pewne ich uporządkowanie hierarchiczne. Uporządkowanie odnosi się do zasad dostępu (zasad komunikacji) komponentów. Źródłem wszelkiej aktywności są równolegle pracujące procesy stanowiące umownie najniższy poziom hierarchii. Ogólnie uporządkowanie to można przedstawić w postaci skierowanego grafu bez pętli (ścieżek zamkniętych): wierzchołki wejściowe tego grafu (te, do których nie prowadzą inne łuki) to procesy, a pozostałe wierzchołki to - w uproszczeniu - monitory; łuki wychodzące z danego komponentu wskazują inne komponenty, do których dany komponent ma dostęp.

Bardziej szczegółowo strukturę programu wyjaśniają podane niżej reguły. Używa się w nich pojęcia atrybutów komponentu programu. Przez atrybuty składowej rozumie się jej parametry oraz zdefiniowane w jej wnętrzu stałe, typy, zmienne, procedury i funkcje.

1. Program zawierający procesy współbieżne ma najbardziej zewnętrzny blok (blok początkowy) spełniający rolę bezimiennego procesu początkowego i inicjalizującego pozostałe zadeklarowane w nim komponenty.

2. Komponent programu i jego parametry aktualne muszą być atrybutami innego komponentu.

3. Komponent może być inicjalizowany tylko jeden raz i musi to być dokonane wewnątrz tego komponentu, w którym dany komponent jest zadeklarowany.

4. Gdy komponent zakończy swe działanie, zadeklarowane w nim zmienne nadal istnieją.

5. Wszystkie składowe dostępne danej składowej muszą być jej atrybutami.

6. Dostęp do zmiennych lokalnych monitora lub klasy, po ich zainicjalizowaniu, może odbywać się wyłącznie przez ich procedury wejściowe. Procedury te są dostępne w komponencie, w którym są zadeklarowane.

7. Dostęp do zmiennych lokalnych procesu dla programów sekwencyjnych, wykonywanych pod ich nadzorem, może odbywać się wyłącznie za pośrednictwem procedur wejściowych monitora.

8. Procedury (zwykle) mają dostęp do własnych atrybutów oraz do atrybutów składowej, wewnątrz której są definiowane.

9. Komponenty mogą używać stałych i typów zdefiniowanych na zewnątrz, w innych komponentach, w których są zanurzone.

3.9. Wybrane własności implementacji języka na minikomputerach PDF11

Concurrent Pascal - jak wspomniano - nie dopuszcza rekursji procedur, natomiast rekursja jest dopuszczalna w programach sekwencyjnych napisanych w Pascalu, a wykonywanych pod kontrolą procesów współbieżnych. Znajduje to odbicie w deklaracji procesów, w której można wskazać dodatkowy obszar pamięci wymagany do wykonania programów sekwencyjnych. Piszac nagłówek

process (<wykaz parametrów>) + <liczba całkowita>;

definiujemy proces, z którym będzie związany obszar pamięci, o pojemności <liczba całkowita> bajtów, przeznaczony na pamiętanie dwóch stosów: jeden (heap) dla zmiennych dynamicznych generowanych procedurą new, drugi (stack) dla zmiennych tymczasowych wywoływanych procedur.

W dalszej części przedstawiono procedury standardowe i bardziej szczegółowo omówiono komunikację z urządzeniami zewnętrznymi.

Funkcja standardowa

attribute (x)

definiuje wartość atrybutu x wołającego procesu. Atrybut x może przyjmować wartości typu

```
type attrindex = (caller, heaptop, propline,
                   progresresult, runtime).
```

attribute (caller) daje w wyniku liczbę całkowitą identyfikującą wołający proces. Procesem są nadawane liczby 0,1,2,... odpowiadające kolejności ich inicjalizacji.

attribute (heaptop) daje liczbę całkowitą określającą adres szczytu stosu służącego do przechowywania zmiennych dynamicznych.

attribute (propline) wywołana po zakończeniu programu sekwencyjnego podaje numer linii tekstu kodu, na której zatrzymał się program.

attribute (result), wywołana w tej sytuacji jak wyżej, daje w wyniku wartość typu

```
type resulttype = (terminated, overflow, pointererror,
                   rangeerror, varianterror, heaplimit, stacklimit).
```

Pierwszy z tych elementów tego typu oznacza poprawne zakończenie programu, a pozostałe - zakończenie błędne, z podaniem przyczyny błędów.

attribute (runtime) określa czas w sekundach, który został przeznaczony przez procesor na wykonanie danego procesu od momentu jego inicjalizacji.

Procedura wait opóźnia wołający proces do momentu wygenerowania kolejnego jednosekundowego sygnału przez zegar systemu.

Funkcja realtime definiuje czas w sekundach, który upłynął od momentu inicjalizacji systemu.

Procedura wejścia/wyjścia ma postać io(x,y,z), gdzie parametr z wskazuje urządzenia peryferyjne, y - rodzaj operacji wejścia/wyjścia, zaś x - zmienną, która w tej operacji uczestniczy. Parametry x,y,z są wzajemnie od siebie uwarunkowane, zależnie od rodzaju urządzenia wejścia/wyjścia. Parametr z musi być typu

```
type żądanie = (typedevice, diskdevice, tapedevice,
                printdevice, carddevice),
```

a parametr y - typu

```
type io param =
  record
    operation : iooperation;
    status : ioresult;
    arg : ioarg
  end,
```

gdzie

```
type iooperation = (input,output, move, control);
type ioresult = (complete, intervention, transmission,
                failure, endfile, endmedium, startmedium).
```


Znaczenie elementów ostatniego typu jest następujące:

- complete - poprawne zakończenie operacji;
- intervention - błędne zakończenie operacji; błąd może być usunięty przez obsługę operatorską (np. brak papieru w drukarce);
- transmission - błąd transferu informacji (np. błąd parzystości), operacja może być powtórzona;
- failure - trwała awaria urządzenia peryferyjnego;
- endfile - osiągnięto znacznik końca pliku (na taśmie lub na dysku);
- endmedium - osiągnięto znacznik końca taśmy magnetycznej;
- startmedium - osiągnięto znacznik początku taśmy magnetycznej.

Wywołanie procedury `io` w procesie powoduje zawieszenie wykonania procesu aż do momentu zakończenia pozytywnego lub negatywnego wskazanej operacji. Szczegóły wykonania tej procedury dla poszczególnych urządzeń peryferyjnych - wartości i znaczenie parametrów procedury - przedstawiają się następująco.

Terminal (typedevice)

`x` jest typu znakowego (char).

`input` wprowadza pojedynczy znak do pamięci, podstawia pod `x`, a jego echo wyświetla na ekranie terminala (znak BELL nie może być wprowadzony),

`output` wyprowadza pojedynczy znak (zawartość `x`) na ekran terminala.

`control` opóźnia proces wywołujący procedurę `io` aż do momentu naciśnięcia klawisza BELL przez operatora.

`status` może przyjąć tylko wartość complete.

`arg` nie jest wykorzystywany.

Dysk (diskdevice)

`x` jest typu stronicowego

`type blacktype = univ array (.1..512.) of char`

`arg` jest nazywany indeksem strony i przyjmuje wartości 0..4799.

`input` (`output`) wprowadza (wyprowadza) stronę o podanym indeksie do (`z`) pamięci operacyjnej `z` (do) pamięci dyskowej.

`control` rozpoczyna wykonanie programu zapamiętanego na dysku z początkiem na stronie wskazanej przez indeks.

`status` przyjmuje wartości complete, intervention, transmission, failure.

Taśma magnetyczna (tapedevice)

`x` jest typu stronicowego jak wyżej.

`input` (`output`) wprowadza (wyprowadza) kolejną stronę (na kolejną stronę) `z` (do) pamięci taśmowej (pamięci operacyjnej) do (`z`) operacyjnej (taśmowej).

move przesuwaa taśmę w sposób uzależniony od wartości arg następująco:

writeeof - wprowadza na taśmę znacznik końca pliku
 rewind - przewija taśmę do początku,
 upspace - przesuwaa taśmę do przodu o jedną stronę,
 backspace - przesuwaa taśmę do tyłu o jedną stronę,
 status - przyjmuje wszystkie wartości typu ioresult.

Drukarka (printdevice)

x jest typu liniowego

type linetype = array (.1..132.) of char

output wyprowadza linię (132 lub mniej znaków zakończonych znakami powrotu karetki, zmiany wiersza lub zmiany strony) na drukarkę,
 status przyjmuje wartości complete lub intervention.

Czytnik kart (carddevice)

x jest typu kartowego

type card = array (.1..80.) of char

output wyprowadza zawartość jednej karty 80 kolumnowej do pamięci,
 status przyjmuje wartość complete intervention, transmission, failure.

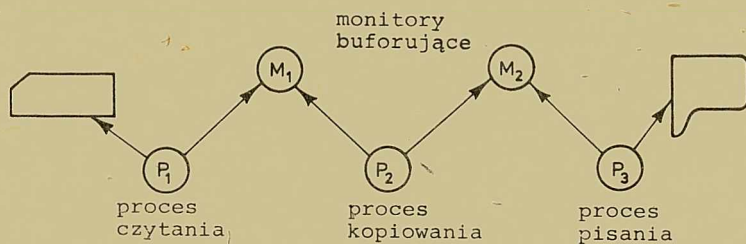
3.10. Program przykładowy

Rozpatrzmy proste zadanie kopiowania zawartości kart na drukarkę wierszową połączone z odpowiednią redakcją (formatowaniem) wydruku.

Najprostsze rozwiązanie zadania, jakie się nasuwa, może być programem sekwencyjnym działającym według schematu:

cycle wprowadź kartę;
 wyprowadź linię
end

Ma ono tę niedogodność, że proces kopiowania jest bardzo wolny ze względu na naprzemienną pracę obu urządzeń peryferyjnych. Efektywniej wykorzystując pracę urządzeń jest takie rozwiązanie, w którym oddzielne, równoległe pracujące procesy obsługują niezależnie oba urządzenia. Dlatego sensowne jest przyjęcie następującej struktury programu (rys. 3.1).



Rys. 3.1. Wstępna struktura programu

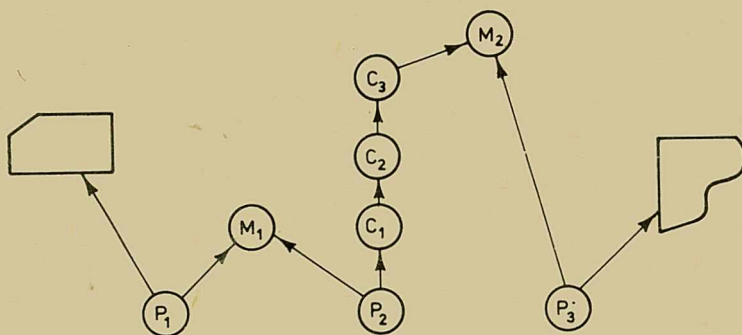
Proces P_1 obsługuje czytnik kart, a zawartość przeczytanej karty przekazuje do monitora buforującego M_1 . Proces P_2 pobiera dane od monitora M_1 , odpowiednio je redaguje i przekazuje do monitora buforującego M_2 . Proces P_3 dane przygotowane w monitorze M_2 wyprowadza na drukarkę wierszową.

Dalsza strukturalizacja programu będzie wynikać z przyjętych zasad redagowania tekstu. Przyjmijmy, że ciąg kart 80-kolumnowych na wejściu składa się z plików poprzedzielanych kartą sterującą, której zawartością jest znak #, po którym następują same spacje. Tekst wyprowadzany na drukarkę jest dzielony na strony zawierające 60 wierszy 132 znakowych. Wymaga się, aby:

- zawartość pojedynczej karty (innej od karty sterującej) była drukowana w jednym wierszu z 24 znakami odstępu z lewej strony jako marginesem,

- na jednej stronie wydruku nie mogą znajdować się wiersze należące do różnych plików; pliki powinny być przedzielone stroną pustą.

Wynika stąd propozycja strukturalizacji programu przez wprowadzenie klas wykonujących etapowo pewne funkcje redakcyjne (rys. 3.2).



Rys. 3.2. Pełna struktura programu

Zadaniem klasy C_1 jest redakcja pliku wyjściowego, w szczególności dołączenie pustych stron przedzielających pliki. Klasa C_2 redaguje strony, uzupełniając w szczególności niepełne strony pustymi liniami. Klasa C_3 redaguje poszczególne wiersze, tworząc lewy i prawy margines dla wydruku.

Przy tak przyjętej strukturze programu można zdefiniować poszczególne jego komponenty. Najpierw zdefiniujemy występujące tu typy systemowe, a po tym przedstawimy cały program.

Procesy

Proces P_1 - jako typ - jest zdefiniowany następująco

```

type proces czytania =
  process (bufor monitor buforujący);
  var param: ioparam;
      tekst, błąd : linia;
      nrznaku: integer;
  begin
    for nrznaku := 1 to 80 do
      błąd (.nrznaku.) := 1 ;
      param. operation := input;
    with param do
      cycle
        repeat io (tekst, param, carddevice)
        until status < > intervention;
        if status < > complete then
          test := błąd
        bufor. przyjmij (tekst)
      end
    end:
  end:

```

Występujący tu typ ioparam jest rozumiany tak, jak w p. 3.9, natomiast linia to typ zdefiniowany przez:

```

type linia = array (.1..132.) of char

```

Proces ma jeden parametr formalny bufor. Zgodnie z zasadami podanymi w p. 3.8 oznacza to, że będzie on miał dostęp do pewnego komponentu (monitora) typu "monitor buforujący". Dostęp ten jest faktycznie wykorzystany w treści procesu, gdzie występuje instrukcja

```

bufor.przyjmij (tekst)

```

stanowiąca wywołanie procedury wejściowej monitora.

Proces P_3 - jako typ - jest "antysymetryczny" w stosunku do P_1 i przedstawia się następująco:

```

type proces pisanie =
  process (bufor: monitor buforujący),
  var param: ioparam
      tekst: linia;
  begin
    param.operation:=output;
  cycle
    bufor.odaj(tekst);
    repeat(tekst,param,printdevice)

```



```

    until param.status=complete;
  end
end

```

Proces P_2 - jako typ - ma postać:

```

type proces kopiowania =
  process (buforwe, buforwy: monitor-buforujący);
  var konsument: redakcja pliku; "klasa" tekst: linia;
begin
  init konsument (buforwy);
  cycle
    buforwe.oddaj(tekst)
    konsument.pisz(tekst)
  end
end

```

Proces ma dwa parametry formalne buforwe, buforwy, z których pierwszy jest jawnie użyty w tekście procesu, natomiast drugi - użyty jako parametr konsumenta (klasy zdefiniowanej dalej) - służy do przekazania praw dostępu nowo kreowanemu komponentowi.

Monitory

Oba monitory M_1 , M_2 programu są wspólnego typu

```

type monitor-buforujący =
  monitor
    var zawartość: linia;
    pełny: boolean;
    odbiorca, nadawca: queue;
  procedura entry oddaj (var tekst: linia);
begin
  if not pełny then delay (odbiorca);
  tekst := zawartość;
  pełny := false;
  continue (nadawca);
end:
  procedure entry przyjmij (tekst: linia);
begin
  if pełny then delay (nadawca);
  zawartość := tekst;
  pełny := true;
  continue (odbiorca);
end:
  "instrukcja inicjalizująca":
begin

```

```

    pełny := false
  end;

```

Monitory przechowują tylko jedną porcję danych (linię); procedury wejściowe służą do wpisu (przyjmij) i odczytu (oddaj) kolejnych linii pomiędzy parą komunikujących się procesów.

Klasy

Jak już wspomniano, definiuje się trzy klasy. Są one wzajemnie w sobie zagnieżdżane

```

type redaktor_pliku =
  class (bufor: monitor_buforujący);
  var konsument: redaktor_strony; "klasa"
      koniecpliku: boolean;
  function dalej (tekst: linia): boolean;
  var nrznaku: integer;
  begin
    if tekst (.1.) <> '#' then dalej := true
    else begin
      nrznaku := 80;
      while tekst (.nrznaku.) = ' ' do
        nrznaku := nrznaku-1;
        dalej := (nrznaku < 1)
      end
    end;
  end;
procedure entry pisz (tekst: linia);
begin
  if koniecpliku then
    begin
      konsument.przeskocz;
      koniecpliku := false
    end;
  if dalej (tekst) then konsument.pisz(tekst)
  else begin
    konsument.przeskocz;
    koniecpliku := true
  end
end;
"instrukcja inicjalizująca"
begin init konsument(bufor);
      koniecpliku := true
end
end

```


Zmienna konsument jest egzemplarzem typu redaktor strony:

```

type redaktor_strony =
  class (bufor: monitor_buforujacy);
  var konsument: redaktor_wiersza; "klasa"
      nrlinii: integer;
  procedure nowa_strona;
  var tekst: linia;
  begin
    tekst(.1.):= ff; "znak zmiany strony"
    konsument.pisz(tekst);
    tekst(.1.):=nl; "znak zmiany wiersza"
    nrlinii := 1
  end;
  procedure entry przeskocz;
  begin nowastrona end;
  procedure entry pisz (tekst: linia);
  begin
    konsument.pisz(tekst);
    if nrlinii = 60 then nowastrona
    else nrlinii := nrlinii+1
  end;
  "instrukcja inicjalizująca"
  begin init konsument (bufor)
      nowa_strona
  end;
end;

```

Wewnątrz zdefiniowanej klasy pozostaje jeszcze nieokreślona, ostatnia już, klasa redaktor linii:

```

type redaktor_linii =
  class (bufor: monitor_buforujacy),
  var obraz:linia;
      nrznaku: integer;
  procedure entry pisz (tekst:linia);
  begin
    for nr znaku := 27 to 106 do
      obraz(.nrznaku.):=tekst(.nrznaku-26.)
      bufor.przyjmij(obraz)
    end
  "instrukcja inicjalizująca"
  begin
    for nrznaku:=1 to 26 do

```

```

        obraz(.nrznaku.):= '—';
        obraz(.107.) := nl "znak nowej linii"
    end;

```

Program

Po opisie typów wszystkich komponentów można napisać pełny tekst programu; wykropkowano w nim te fragmenty, które zostały uprzednio zdefiniowane.

```

"xxxxx Kompletny program xxxxx"
type iodevice = (typedevice, diskdevice,
    tapedevice, printdevice, carddevice);
type iooperation = (input, output, move,
    control);
type ioresult = (complete, intervention, transmission,
    failure, endfile, endmedium, startmedium);
type loparam = record
    operation : iooperation;
    status : ioresult;
    arg : integer
end:
type linia = array (.1.132.) of char;
const nl = '(:10:)' ; ff = '(:12:)' ;
"nl - nowa linia, ff - nowa strona"
type monitor_buforujący =
    monitor ..... end;
type redaktor_linii =
    class (bufor: monitor_buforujący);
    .....
end;
type redaktor_strony =
    class (bufor: monitor_buforujący);
    .....
end;
type redaktor_pliku =
    class (bufor: monitor_buforujący);
    .....
end;
type process_czytania =
    process (bufor: monitor_buforujący);
    .....
end;
type process_kopiowania =

```



```

process (buforwe, buforwy: monitor_buforujacy);
.....
end;
type proces_pisania =
process (bufor: monitor_buforujacy);
.....
end;
var buforwe, buforwy: wspólny_bufor;
    czytnik: proces_czytania;
    kopista: proces_kopiowania;
    pisak: proces_pisania;
"treść bloku początkowego programu"
begin
    init buforwe, buforwy,
        czytnik (buforwe),
        kopista (buforwe, buforwy),
        pisak (buforwy)
end.

```

Treścią bloku początkowego programu jest inicjalizacja monitorów i procesów programu. Instrukcja init obejmuje wszystkie występujące po niej nazwy komponentów; inicjalizacja odbywa się w kolejności występowania nazw. Należy zwrócić uwagę, że inicjalizacja odbywa się w takiej kolejności, iż w momencie inicjalizacji danego komponentu zainicjalizowane są już komponenty będące jego parametrami aktualnymi.

3.11. Uwagi końcowe

Język Concurrent Pascal definiuje program jako hierarchicznie uporządkowaną strukturę swych komponentów. Struktura ta jest statyczna, dzięki czemu już na etapie kompilacji można wykryć wszelkie niezgodności z przyjętymi prawami dostępu. Przy zachowaniu praw dostępu i hierarchicznej strukturze unika się możliwości powstania trwałych blokad podczas realizacji programu. Cechy te są jednocześnie przyczyną pewnych kłopotów.

Pierwszy z nich może objawiać się w małej efektywności współpracy procesów. Chodzi tu o sytuację, gdy dwa procesy komunikują się poprzez strukturę wzajemnie się wywołujących monitorów; wtedy wywołanie jednego monitora może blokować dostęp do innych monitorów. Istnieje wiele propozycji poprawienia tej sytuacji, np. [38].

Drugi kłopot to niemożność dynamicznej generacji komponentów, co jest utrudnieniem w wielu zastosowaniach. Stąd propozycja pewnych rozszerzeń języka, np. język Pascal - Plus [64].

Przyjęte prawa dostępu wprowadzają pewne utrudnienia w analizie własności programów; pewnym rozwiązaniem tych trudności jest koncepcja warstwy, jako dodatkowej konstrukcji językowej [62].

Niektórzy autorzy wyrażają pogląd o braku praktycznej potrzeby tworzenia wielu egzemplarzy typów systemowych [15] itd.

Przytoczone uwagi wskazują na duże zainteresowanie językiem i różne próby jego zastosowań. Na szczególne podkreślenie zasługuje napisanie i uruchomienie kilku systemów operacyjnych [10]. Niezależnie od swych słabości, Concurrent Pascal stanowi znaczący (w tej chwili już historyczny) krok w rozwoju programowania współbieżnego.

ĆWICZENIA

1. Zaprojektować w języku Concurrent Pascal szkielecik małego systemu operacyjnego przeznaczonego do wykonywania wsadu programów napisanych w Pascalu. Założyć, że programy sekwencyjne w Pascalu mogą korzystać z procedur standardowych (p. 3.9) wyłącznie za pośrednictwem projektowanego systemu operacyjnego.

2. Przykładowy program, przedstawiony w p. 3.10, rozszerzyć w taki sposób, aby o wszelkich awariach czytnika kart lub drukarki wierszowej informować operatora za pośrednictwem terminala.

3. Założyć, że procesor ma spełniać rolę węzła podsieci teletransmisyjnej. Oznacza to, że do procesora jest podłączona pewna liczba linii wejściowych i linii wyjściowych. Liniami napływają z ustaloną szybkością ciągi znaków. Znaki te są buforowane pojedynczo w tzw. adapterach liniowych. Pewne podciągi tych znaków, wyróżnione pewnym znakiem początkowym i końcowym, tworzą pakiety. Pakiety są w pewien sposób przetwarzane, a następnie kierowane w linie wyjściowe. Zaproponuj instrukcję wejścia/wyjścia komunikacji z adapterem liniowym. Zakładając, że znane są procedury przetwarzania pakietów w języku Concurrent Pascal, zaprojektuj strukturę systemu operacyjnego węzła. Szczególną uwagę należy zwrócić na konieczność utrzymania reżimów czasowych, aby nie wystąpiło "gubienie" odbieranych lub wysyłanych znaków.

4. Założyć, że dysponujemy pamięcią dyskową składającą się z 24 cylindrów na każdej z dwóch powierzchni po 12 stron (512 znaków). Obowiązuje przy tym następujący algorytm zamiany argumentu (nr-strony) w procedurze wejścia/wyjścia, na adres fizyczny:

powierzchnia := nr_strony div 12 mod 2;

cylinder := nr_strony div 24;

sektor := nr_strony mod 12;

Zaprojektować w języku Concurrent Pascal system organizujący wirtualną pamięć plików na dysku.

5. Napisać w języku Concurrent Pascal program symulujący działanie czytników i pisarzy (ćw. 1, rozdz. 2).

6. Napisać w języku Concurrent Pascal program symulujący działanie filozofów (ćw. 5, rozdz. 2).

7. Rozważmy zagadnienie przydziału m jednorodnych zasobów zwrotnych pomiędzy procesy współbieżne. Algorytm bankiera, który rozwiązuje to zagadnienie w taki sposób, aby uniknąć zastoju, działa następująco. Zakłada się, że dla każdego procesu jest znana maksymalna liczba jednostek zasobów, którą można jednocześnie wykorzystać. Przydział procesowi maksymalnej liczby zasobów powoduje, po upływie skończonego odcinka czasu, zwrot wszystkich zasobów. Przydział części maksymalnej liczby zasobów danemu procesowi może nastąpić tylko wówczas, gdy pozostała pula wolnych zasobów jest wystarczająca do zaspokojenia w pełni żądań przynajmniej jednego procesu. Napisać w języku Concurrent Pascal fragment programu przydzielający procesom zasoby zgodnie z podanym wyżej algorytmem. (Zwrócić uwagę na fakt, aby w chwili zwolnienia zasobów przez pewien proces zostały odwieszone wszystkie oczekujące procesy, których ządania mogą być zaspokojone.)

8. Pewien monitor w Concurrent Pascalu ma zawierać trzy procedury wejściowe EP1, EP2, EP3. Treścią tych procedur jest wywołanie pewnych lokalnych procedur monitora - odpowiednio P1, P2, P3. Wywołanie i wykonanie procedur może jednak nastąpić pod warunkiem, że są spełnione pewne warunki logiczne zdefiniowane przez lokalne funkcje boolowskie - odpowiednio B1, B2, B3. Napisać szkielet takiego monitora, który zachowując opisane zasady, umożliwia współpracę grupie n procesów wywołujących jego procedury wejściowe. Szeregowanie zawieszonych procesów powinno odbywać się wg zasady "proces najrzadziej przetwarzany w monitorze ma pierwszeństwo".

4. JEZYK PROGRAMOWANIA MODULA

4.1. Wprowadzenie

Język Modula (Modular Language), podobnie jak Concurrent Pascal, należy do klasy języków bazujących na założeniu wspólnego środowiska dla równolegle wykonywanych procesów. Język opracował N. Wirth z Uniwersytetu Technicznego w Zurychu (ETH - Eidgenössische Technische Hochschule) w latach 1974-76. Pierwszy kompilator tego języka (napisany niemal w całości w Pascalu i uruchomiony na CDC 6100) w eksperymentalnej wersji na minikomputer PDP11/45 ukończono w 1976 r. [67, 68, 69]. Jest on także oparty na języku Pascal i - zdaniem autora - okazał się, w zestawieniu z Pascalem, językiem znacznie prostszym w implementacji.

Zasadniczym przeznaczeniem języka Modula jest programowanie sterowania procesami technologicznymi. W zakresie sterowania takimi procesami stawiane są dwa wymagania: potrzeba mechanizmów obliczeń równoległych oraz mechanizmów zarządzania urządzeniami peryferyjnymi. Zasadniczą przeszkodą we wprowadzeniu takich mechanizmów, zwłaszcza w zakresie zarządzania urządzeniami peryferyjnymi, jest duża zależność od cech architektury, a nawet konfiguracji systemów komputerowych. Praktycznym rozwiązaniem jest zaakceptowanie takiej sytuacji i wprowadzenie do języka konstrukcji, które przesłonią obiekty zależne maszynowo, tzn. ograniczą ich znaczenie lub obecność do specyficznego, zwykle małego fragmentu programu (device module).

Podstawowymi konstrukcjami języka są: procesy (process), moduły (module), moduły pośredniczące (interface module) oraz moduły peryferyjne (device module). Po porównaniu tych konstrukcji z odpowiednimi konstrukcjami Concurrent Pascala można stwierdzić, że pojęcia procesów są podobne, odpowiednikiem modułu jest klasa, odpowiednikiem modułu pośredniczącego zaś - monitor, natomiast moduł peryferyjny nie ma swego odpowiednika. Istotną różnicą w stosunku do Concurrent Pascala jest to, że Modula nie wprowadza pojęcia typów systemowych; w związku z tym moduły są definiowane jako pojedyncze egzemplarze.

Ze względu na pewne podobieństwa do języka Concurrent Pascal przedstawiony dalej opis języka Modula jest bardzo zwarty; interpretację wielu szczegółów można wywnioskować z przedstawionych przykładów.

4.2. Mechanizmy sekwencyjne

Elementami alfabetu języka są znaki należące do kodu ASCII, identyfikatory liczby, łańcuchy (ciągi znaków zamknięte pomiędzy dwoma apostrofami) oraz operatory i ograniczniki:

+	(	<u>div</u>	<u>until</u>	<u>const</u>
-	)	<u>mod</u>	<u>while</u>	<u>var</u>
*	[	<u>or</u>	<u>do</u>	<u>type</u>
/	]	<u>and</u>	<u>loop</u>	<u>array</u>
=	.	<u>not</u>	<u>when</u>	<u>record</u>
<>	,	<u>if</u>	<u>exit</u>	<u>procedure</u>
<	;	<u>they</u>	<u>begin</u>	<u>process</u>
<=	:	<u>elseif</u>	<u>and</u>	<u>modula</u>
<	⌈	<u>else</u>	<u>with</u>	<u>interface</u>
>=		<u>case</u>	<u>value</u>	<u>device</u>
	(*	<u>of</u>	<u>xor</u>	<u>use</u>
:=	*)	<u>repeat</u>		<u>define</u>

Symbole (* oraz *) są nawiasami dla komentarzy.

Struktury danych w Moduli są podzbiorem struktur Pascala. Typami prostymi Moduli są: typ całkowity (integer), logiczny (Boolean), znakowy (char), wyliczeniowy oraz bitowy (bits). Ten ostatni jest standardową tablicą boolowską

array 0: w of Boolean,

gdzie w stanowi parametr implementacyjny, dla PDP11 w = 15. (Uwaga: nieco inna notacja granic indeksu tablicy.)

Na zmiennych bitowych można wykonywać operacje logiczne and, or, xor, not. Wymienione operacje są wykonywane na wszystkich odpowiadających sobie elementach operandów. Typ bitowy zastępuje typ zbiorowy z Pascala.

Typami złożonymi są tylko tablice (array) oraz rekordy (record) bez wariantów.

Deklaracje stałych, typów, zmiennych są również takie jak w Pascalu.

Instrukcje zawierają bogaty zestaw struktur sterowania, różniący się nieco od struktur sterowania w Pascalu. Jedną z tych różnic dotyczy syntaktyki: przyjęto bowiem zasadę, że każda instrukcja nie tylko zaczyna się, ale również kończy symbolem nawiasu - nawiasem kończącym jest słowo end. Instrukcja alternatywy (if) może więc mieć postać:

```
if B then S
{ elseif B' then S' }
[ else S" ] end,
```

gdzie B, B' są wyrażeniami logicznymi, S, S', S'' dowolnymi ciągami instrukcji. Stosowane tutaj i dalej oznaczenia pomocnicze $\{ \}$ oraz $[]$ mają sens następujący:

$\{E\}$ oznacza zbiór napisów złożony z napisu pustego oraz napisów $E, EE, EEE, \dots$

$[E]$ oznacza zbiór złożony z napisu pustego oraz napisu E .

Instrukcja wyboru (case), iteracyjna (repeat) oraz wiążące (with) mają w Pascalu nawias kończący end i dlatego taką samą postać mają w Moduli. Instrukcja iteracyjna (while) ma dodatkowy nawias kończący

while B do S end.

Modula ma jeszcze jedną instrukcję iteracyjną (loop) w postaci:

```
loop  $S$  when  $B$  [do  $S'$ ] exit
       $S''$  end
```

Stąd ogólną postacią tej instrukcji jest:

```
loop  $S_1$  when  $B_1$  do  $X_1$  exit
       $S_2$  when  $B_2$  do  $X_2$  exit
      .....
       $S_n$  when  $S_n$  do  $X_n$  exit
       $S$ 
end
```

Wykonanie instrukcji pętli przebiega następująco: Najpierw wykonuje się ciąg instrukcji S_1 , a następnie oblicza się warunek logiczny B_1 . Jeżeli jest on prawdziwy, oblicza się ciąg instrukcji X_1 , po czym obliczenia pętli kończą się; jeżeli jest on fałszywy, oblicza się ciąg S_2 itd. Po wykonaniu ciągu S wykonanie pętli powtarza się od ponownego wykonania S_1 . Instrukcja pętli (loop) jest bardzo ogólna, jej przypadkami szczególnymi są poprzednio omówione instrukcje iteracyjne.

Konstrukcja procedury w Moduli jest w stosunku do procedur w Concurrent Pascalu, a nawet w Pascalu, bardziej rozbudowana. Deklaracja procedury - podobnie jak w Pascalu - składa się z nagłówka z identyfikatorem i listy parametrów formalnych oraz z bloku, stanowiącego treść procedury. Również, jak w Pascalu, wyróżnia się procedury właściwe oraz funkcyjne, przy czym rozróżnienie pomiędzy nimi polega tylko na tym, że nagłówek procedury funkcyjnej jest zakończony typem wyniku procedury. Przykładowo deklaracja procedury funkcyjnej obliczającej największy wspólny podzielnik ma postać:

```
procedure nwp ( $x, y$ :integer):integer;
  var  $a, b$ : integer; (* zakłada się  $x, y > 0$  *)
  begin  $a := x$ ;  $b := y$ ;
```



```

while a < > b do
  if a < b then b := b - a else a := a - b
end
end
nwp := a
end nwp

```

W treści procedury funkcyjnej musi wystąpić instrukcja podstawienia wartości pod nazwą tej procedury. Każda procedura musi kończyć się słowem end, po którym jest powtórzony jej identyfikator.

Komunikacja procedury przez parametry odbywa się dwoma znanymi sposobami: wołanie przez zmienną (parametry formalne specyfikowane przez var) oraz przez wartość (parametry formalne niespecyfikowane lub specyfikowane przez const).

W bloku stanowiącym treść procedury mogą być deklarowane obiekty lokalne stałe, typy, zmienne, procedury i moduły (p. dalej). Procedury mogą więc być zagnieżdżane, a także mogą wywoływać się rekursywnie.

Procedura ma dostęp do swych obiektów lokalnych oraz parametrów, może także istnieć dostęp do obiektów nielokalnych (nie tylko do zmiennych, jak w Pascalu). Takie obiekty nielokalne muszą być także wyspecyfikowane w nagłówku procedury w postaci listy obiektów importowanych (use list). Lista ta stanowi element mechanizmu definiowania układu odniesienia [62] omówionego dalej przy konstrukcji modułu.

Standardowe procedury składają się z procedur właściwych:

inc (x,n)	równoważnej	x := x + n
dec (x,n)	"	x := x - n
inc(x)	"	x := x + 1
dec(x)	"	x := x - 1
halt	"	zakończenie programu

i procedur funkcyjnych

off(b1,b2)	równoważnej	b1 <u>end</u> b2 = [], gdzie b1, b2 są typu bitowego, [] ciąg bitów złożony z samych zer,
off(b)	równoważnej	b = []
among(i,b)	"	b[i]; b - wyrażenie bitowe
low(a)	"	dolnej wartości indeksu tablicy a
high(a)	"	górnej wartości indeksu tablicy a
adr(v)	"	adresowi zmiennej v
size(v)	"	rozmiarowi zmiennej v
integer(x)	"	liczbie porządkowej x w zbiorze wartości typu, do którego należy x,
char(x)	"	znakowi o liczbie porządkowej x.

Moduł jest najbardziej ogólną konstrukcją w Moduli stanowiącą uogólnienie pojęcia procedury i bloku. Moduł stanowi kolekcję stałych, typów, zmiennych i procedur. Są one atrybutami modułu i rozpoczynają swe istnienie w chwili wywołania procedury względem której moduł jest lokalny. (Jak wspomniano przy opisie procedur, jej obiektami lokalnymi mogą być m.in. moduły.) Ponieważ cały program jest także modułem, atrybuty modułu - programu rozpoczynają swe istnienie w chwili startu programu.

Istotną cechą modułu, stanowiącego "ogrodzenie" dookoła pewnej grupy obiektów, jest precyzyjne zdefiniowanie zasad dostępu do tych obiektów oraz zasad dostępu wewnątrz modułu do obiektów nie będących jego atrybutami - obiektów zdefiniowanych na zewnątrz modułów. W swoim nagłówku moduł ma dwie listy identyfikatorów. Pierwsza lista obiektów eksportowanych (specyfikowana słowem define) wymienia te wszystkie obiekty modułu, które są dostępne na zewnątrz. Druga lista obiektów importowanych (specyfikowana słowem use) wymienia te obiekty zadeklarowane na zewnątrz modułu, które są dostępne w jego wnętrzu. W ten sposób wyraźnie oddziela się te obiekty, które abstrakcyjnie reprezentują działanie modułu, od tych, które stanowią implementację takiego działania. Moduł przesłania swe wnętrze, które jest nieistotne dla pozostałej części programu lub to, które musi być chronione przed niepożądanym dostępem. Moduły mogą być zagnieżdżane. Rozpatrzmy przykład

```

module M1;
  define b,c;
  use a;
  deklaracja d
module M2;
  define c,e;
  use d;
  deklaracja c,e,f
  c,d,e,f są dostępne w tym miejscu
end M2;
procedure b;
  deklaracja f
  a,b,c,d,e,f są dostępne w tym miejscu
end b;
  a,b,c,d,e są dostępne w tym miejscu
end M1

```

Obiekt *a* jest zdefiniowany na zewnątrz modułu *M1*, zaś *b,c* są obiektami zdefiniowanymi wewnątrz. Podobną rolę odgrywają *d* oraz *c,e* dla modułu *M2*. Wewnątrz procedury *b* są dostępne jej atrybuty, tzn. obiekty *b,f* oraz wszystkie atrybuty dostępne wewnątrz modułu *M1*, tzn. obiekty *a,c*,

d,e. Dostęp ten wynika z normalnych reguł dostępu, takich jakie obowiązują w Pascalu. Gdyby w procedurze występowała lista obiektów importowanych (use), wtedy - poza atrybutami lokalnymi - byłyby dostępne w jej wnętrzu tylko obiekty z tej listy.

Jeżeli obiektem eksportowanym jest typ, to na zewnątrz modułu znany jest tylko identyfikator tego typu, nieznanie natomiast pozostają jego szczegóły strukturalne. A więc zmienne takiego typu mogą być modyfikowane tylko przez procedury tego samego modułu. Jeżeli obiektem eksportowanym jest zmienna, to jej wartość na zewnątrz modułu może być tylko odczytywana.

4.3. Mechanizmy równoległe

Procesy (process), moduły pośredniczące (interface module) oraz mechanizmy synchronizacji są zasadniczymi konstrukcjami równoległymi w języku Modula.

Procesy syntaktycznie podobne są do procedur, lecz ich wywołanie w programie - inaczej niż w przypadkach procedur - rozpoczyna obliczenia przebiegające równoległe z programem, który zainicjował proces. W chwili, gdy sterowanie osiąga koniec procesu, proces ten znika (podobnie jak procedura). Deklaracja procesu ma postać:

```
process <identyfikator>(<lista parametrów formalnych>);  
    <blok>
```

Proces może być deklarowany jako lokalny obiekt procedury, deklaracje procesów nie mogą być zagnieżdżane. Komunikacja procesu z programem poprzez parametry odbywa się tak samo, jak w przypadku procedury.

Proces jest kreowany przez instrukcję inicjalizacji, o takiej samej postaci jak wywołanie procedury. Procesy mogą być kreowane tylko w programie głównym, tzn. nie mogą być kreowane wewnątrz procesów ani wewnątrz procedur. Wielokrotne powtórzenie instrukcji inicjalizacji powoduje utworzenie wielu egzemplarzy tego samego procesu.

Procesy komunikują się ze sobą przez wspólne zmienne, co wymaga mechanizmu wyłącznego dostępu do sekcji krytycznej. Rolę tę spełnia specjalny rodzaj modułu, zwany modułem pośredniczącym (interface module), stanowiący odpowiednik monitora w Concurrent Pascalu.

Moduł pośredniczący udostępnia procesom dostęp do swych zmiennych lokalnych jedynie za pośrednictwem swych procedur, które znajdują się na liście obiektów eksportowanych. Podobnie jak monitory w Concurrent Pascalu, moduły pośredniczące spełniają funkcje synchronizujące poprzez dwa mechanizmy: szeregowania krótkoterminowego i średnioterminowego. Pierwszy z tych mechanizmów jest taki sam jak w Concurrent Pascalu,

drugi natomiast jest nieco inny i opiera się na tzw. sygnałach synchronizujących (odpowiednikach zmiennych kolejkowych).

Sygnały są deklarowane wewnątrz modułu pośredniczącego jako zmienne typu `signal`. Sygnały mogą być wysyłane przez procesy oraz procesy mogą oczekiwać na odbiór sygnału. Na sygnałach można wykonywać następujące operacje - procedury standardowe: `wait`, `send`, `awaited`.

Wywołanie procedury `wait (s,r)` opóźnia proces aż do momentu otrzymania sygnału `s`. Procesowi jest nadana ranga opóźnienia `r` - dodatnia liczba-całkowita, `wait(s)` jest skrótem `wait(s,1)`.

Wywołanie procedury `send(s)` przesyła sygnał `s` do tego procesu, który oczekuje na `s` i ma najwyższą rangę opóźnienia. Jeżeli kilka procesów o tej samej randze oczekuje na `s`, to otrzymuje go ten proces, który był najdłużej opóźniany. Jeżeli żaden proces nie oczekuje `s`, to `send(s)` nie ma żadnego efektu.

Boolowska procedura funkcyjna `awaited(s)` przyjmuje wartość `true` wówczas, gdy co najmniej jeden proces oczekuje na `s`, w wypadku przeciwnym przyjmuje wartość `false`.

Gdy proces wywoła procedurę `wait` wewnątrz modułu pośredniczącego (dokładniej wewnątrz procedury eksportowej tego modułu), zostaje wówczas zawieszony, a moduł pośredniczący staje się wolny dla innych procesów. Jeżeli proces wywoła procedurę `send` wewnątrz modułu pośredniczącego i jeżeli sygnał zostanie przesłany do procesu, który na niego oczekuje, to proces odbierający sygnał zostaje odwieszony i otrzymuje dostęp do modułu, proces zaś nadający sygnał zostaje zawieszony do czasu zakończenia procedury modułu dla odwieszonego procesu. Procedury `wait` i `send` stanowią analogony procedur `delay` i `continue` w Concurrent Pascalu.

Jeżeli zmienna sygnałowa jest eksportowana na zewnątrz modułu pośredniczącego, to operacja `send` nie może być wykonywana poza tym modułem.

Poniżej przedstawiono prosty przykład modułu pośredniczącego ze zmienną sygnałową. Moduł symuluje prosty semafor binarny:

```
interface module rezerwacja zasobu;
  define semafor, P,V, init.;
  type semafor = record
    zajęty: Boolean
    zwolnij: signal
  end;
  procedure P (var S : semafor);
  begin if s. zajęty then wait (s.zwolnij) end;
    s. wolny := true
  end P;
```



```

procedure V (var s: semafor);
begin s. zajęty := false;
      send (s.zwolnij)
end V;
procedure init (var s: semafor);
begin s. zajęty := false;
end init;
end rezerwacja zasobu

```

4.4. Własności implementacji na minikomputerach PDP11

Charakterystyczną cechą implementacji języka na PDP11 jest sposób organizacji komunikacji z urządzeniami zewnętrznymi. Wyróżnia się specjalny rodzaj procesów - procesy sterowniki (driver processes) - które sterują pracą urządzeń zewnętrznych. W odróżnieniu od zwykłych procesów, które nie mogą być deklarowane wewnątrz modułów pośredniczących, procesy sterowniki są deklarowane w specjalnym rodzaju modułów pośredniczących zwanych modułami sterownikami (driver module). Właśnie ze względu na swe wyróżnione miejsce deklaracji procesy sterowniki deklarowane są tak jak zwykłe procesy (process). Wyłącznie procesy sterowniki mogą wywoływać procedurę wejścia/wyjścia - doio. Na okres wykonywania tej procedury proces traci wyłączny dostęp do zmiennych lokalnych modułu sterownika (podobnie jak w wypadku wykonywania wait oraz send). Procedura doio stanowi fragment procesu sterownika i zwykle jest poprzedzona instrukcją inicjalizującą urządzenia peryferyjne przez odpowiednie ustawienie zawartości rejestru sterującego urządzenie.

PDP11 ma system priorytetowy i zgodnie z nim przerwania od urządzeń mogą być odpowiednio maskowane i buforowane. Nagłówek modułu sterownika definiuje poziom priorytetowy urządzenia; może on wynosić 4,5 lub 6. (PDP11 jest systemem jednoprosesorowym. Wieloprogramowość uzyskuje się przez podział czasu procesora. Podział wykorzystuje system przerw i luki w czasie procesora powodowane autonomicznym wykonywaniem się operacji wejścia/wyjścia.) Jeżeli proces sterownika wysyła sygnał do procesu o niższym priorytecie, to jest on kontynuowany nadal aż do napotkania procedury wait lub doio. Stanowi to odstępstwo od poprzednio omówionej zasady wykonywania operacji na sygnałach. Normalne procesy mają priorytet 0.

Wszystkie procesy sterowniki są związane z tzw. wektorem przerw. Wektor przerw służy do zapamiętania stanu przerwanej i zawieszonych procesów. Adres tego wektora dla danego procesu jest zapisany w kwadratowych nawiasach w nagłówku procesu.

Procesy sterowniki muszą spełniać następujące ograniczenia:

- nie mogą wysyłać sygnałów do innych procesów sterowników;
- nie mogą wywoływać nielokalnych procedur;
- mogą być aktywowane tylko w jednym egzemplarzu;
- w wywoływaniu procedury wait nie potrzeba definiować rangi opóźnienia.

Moduły sterowniki służą przedstawianiu w możliwie czytelnej formie niezbędnych szczegółów architektury systemu PDP11. W szczególności moduły te wprowadzają deklarację rzeczywistych rejestrów urządzeń. Są one wyróżnione w ten sposób, że jest im przypisywany identyfikator jednocześnie z ich adresem fizycznym pisanym obok identyfikatora w kwadratowych nawiasach.

Przykładowo, podany niżej moduł sterownik definiuje dwie procedury czytajznak, piszznak, które wprowadzają/wyprowadzają pojedynczy znak z/na dalekopis.

```

device module dalekopis [4]; (* 4 - poziom priorytetu *)
define czytajznak, piszznak;
const n = 64; (* rozmiar bufora *)
var SKL [177560B]:bits; (* rejestr statusu klawiatury,
    177560B adres ósemkowy tego rejestru *)
    BKL [177562B]:bits; (* bufor klawiatury *)
    SDRI [177564B]:bits; (* status drukarki *)
    BDR [177566B]:bits; (* bufor drukarki *)
    do_1, do_2, z1, z2; integer;
    niepełny_1, niepełny_2, niepusty_1, niepusty_2:
        :signal;
    buf1, buf2: array 1:n of char;
procedure czytajznak (var znak: char);
begin
    if n1=0 then wait (niepusty1) end;
    znak:=buf1[z1]; z1:=(z1 mod n) + 1;
    dec(n1); send(niepełny1)
end czytajznak;
procedure piszznak (znak: char);
begin
    if n2=n then wait (niepełny2) end;
    buf2[do2]:= znak;
    do2:=(do2 mod n) + 1
    inc(n2); send (niepusty2)
and piszznak;
process sterowanieklawiatyry [60B];

```



```

(* 60B ósemkowy adres wektora przerwania *)
begin
  loop
    if n1=n then wait (niepełny1) end;
    SKL[6] := true; doio; SKL[6] := false;
    buf1 [do1] := BKL;
    do1 := (do1 mod n) + 1;
    inc(n1); send (niepusty1)
  end;
end sterowanieklawiatury;
process sterowanie_drukiem [64B];
begin
  loop
    if n2=0 then wait(niepusty2) end;
    BDR:=buf2 [z2]; z2:=(z2 mod n)+1;
    SDR[6] := true; SDR[6] := false;
    dec(n2); send(niepełny2)
  end
end sterowaniedrukiem
begin
  do1:=1; do2:=1; z1:=1; z2:=1;
  n1:=0; n2:=0
  sterowanieklawiatury; (* kreowanie procesu *)
  sterowaniedrukiem; (* kreowanie procesu *)
end dalekopis;

```

Kolejny moduł sterownik definiuje zmienną czas, której wartość jest automatycznie zwiększana o 1, co każde 20 ms, przez zegar systemu. Sygnał tiktak jest wysyłany co 20 ms, a procedura pauza(n) powoduje opóźnienie wywołującego ją procesu o $n \times 20$ ms.

```

device module czasrzeczywisty[6];
(* 6 - poziom priorytetu *)
define czas, tiktak, pauza;
var czas; integer;
    tiktak: signal;
    SZF[177546B]: bits;
    (* status zegara fizycznego *)
procedure pauza (n:integer);
  var opóźnienie: integer;
  begin opóźnienie := n;
    while opóźnienie > 0 do
      wait (tiktak); dec (opóźnienie)
    end

```

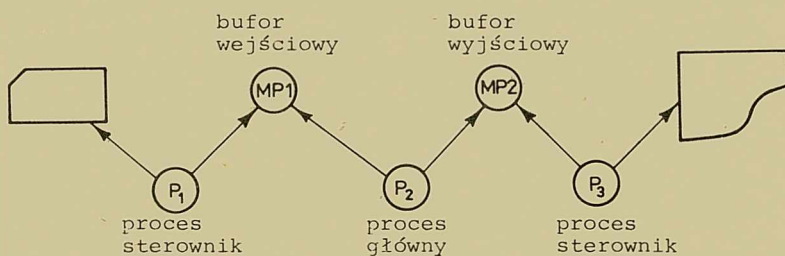
```

    end
  end pauza;
process zegar [100B];
begin SZF[6] := true
  loop doio; inc (czas);
    while a waited (tiktak) do
      send (tiktak)
    end
  end
end zegar;
begin czas := 0; zegar
end czasrzeczywisty:

```

4.5. Program przykładowy

Rozpatrzmy sytuację podobną do tej, która stanowiła podstawę do tworzenia przykładowego programu w języku Concurrent Pascal (p. 3.10). Zagadnienie polega na sterowaniu strumieniem danych z czytnika kart na drukarkę.



Rys. 4.1. Struktura wstępna programu

Nasuującą się strukturę programu pokazano na rys. 4.1, gdzie MP1, MP2 są modułami pośredniczącymi, P1, P3 - procesami sterownikami, P2 - zwykłym procesem. Dalsze uszczegółowienia wynikają z następujących ograniczeń dotyczących pracy urządzeń peryferyjnych.

1. Drukarka używa 7-bitowego kodu ASCII innego od kodu czytnika kart; wynika stąd potrzeba konwersji kodu.

2. Czytnik jednorazowo czyta całą kartę.

3. Ze względu na dużą szybkość transmisji na drukarce sygnały synchronizujące są przesyłane po bloku znaków (po całej linii), zamiast po każdym pojedynczym znaku.

4. Status czytnika kart musi być badany przed i po każdej transmisji. Jeżeli czytnik nie jest gotowy, ponowne badania jego stanu muszą

być periodycznie powtarzane, gdyż czytnik nie wysyła samodzielnie sygnałów o swej gotowości.

```

module strumieniany; (* program *)
const lf=12C; ff=14C; cr=15C;
(* lf,ff,cr odpowiadają znakom sterującym zmiany
linii, zmiany strony i powrotu karetki; 12C,14C,15C
są ósemkowymi liczbami porządkowymi tych znaków *)
var crsign:signal;
device module uplywczasu[6];
    define tiktak:signal;
        var szf[177546B]:bits;
        (* status zegara fizycznego *)
    process sterownikzegara[100B];
    begin szf[6]:= true;
        loop doio; send(tiktak) end
    end sterownikzegara;
device module czytnikkart[6];
    define czytaj;
    use crsign;
    const n=256; (* rozmiar bufora wejściowego *)
    var dox,zx,lp,lz: integer;
        niepełny, niepusty: signal;
        buf: array 1:n of integer;
    procedure czytaj (var x : integer);
    begin dec(lz);
        if lz < 0 then wait (niepusty) end;
        x:=buf zx ; zx:=(z x mod n)+1;
        inc(lp); if lp >= 0 then send (niepełny) end
    end czytaj;
    process sterownikczytnika[230B];
    const m=81; (* rozmiar bloku *)
    var sck[177160B]:bits; (* status czytnika kart *)
        bck[177164B]:integer, (* bufor czytnika kart *)
    procedure wstaw(x:integer);
    begin buf dox :=x; zx:=(zx mod n)+1;
        inc(lz)
    end wstaw;
    begin
        loop dec(lp,m);
            if lp < 0 then wait (niepełny) end;
            while not off (sek,[8,9]) do wait(crsign) end;

```

```

(* procedura off testuje pozycje 8,9 rejestru
sck, jeżeli te obie pozycje nie są jedynkami, to
procedura przyjmuje wartość true *)
sck:= 0,6 ; (* start pracy czytnika *)
loop doio;
    when not off (sck,[14,15]) exit
    wstaw (bck)
end;
wstaw (-1); sck[6]:= false;
(* znak końca linii *)
if lz >= 0 then send (niepusty) end
end
end sterownikczytnika;
device module drukarka [4];
define pisz, piskonlinii, testkolejności;
use if;
const n = 512; (* rozmiar bufora wyjściowego *)
    dc3=23C; dc4=24C;
    (* znaki sterujące drukarki *)
    opóźnienie = 250; (* 10 sec *)
var dox,zx,lp,lz,dal: integer;
    niepełny, niepusty, dozór: signal;
    buf: array 1 : n of char;
    sdr[177514B]: bits; (* status drukarki *)
    bdr[177515B]: integer; (* bufor drukarki *)
procedure pisz (znak:char);
begin dec(lp);
    if ln < 0 then wait (niepełny) end;
    buf[dox] := znak; dox := (dox mod n)+1
end pisz;
procedure piskonlinii;
begin inc(lz); send(niepusty)
end piskonlinii;
procedure testkolejności;
begin
    if lz >= 0 then wait (dozór)
    else dec (del);
    if del = 0 then
        bdr:=dc4; wait (dozór)
    end
end

```



```

end testkolejności;
process sterownik drukarki [200B];
var znak:char;
begin bdr:=dc3;
  loop dec(lz);
    if lz < 0 then
      send (dozór); wait (niepusty)
      bdr:=dc3; del:=opóźnienie
    end;
    repeat znak:=buf [zx], zx:=(zx mod n)+1;
      inc(lz); bdr:=znak;
      if not sdr[7] then
        sdr[6] := true; doio; sdr[6] := false
      end
    until znak:=if;
  if lp >= 0 then send (niepełny) end
end
end sterownikdrukarki;
process strumień;
use czytaj, pisz, pishkonlinii;
const eoi=37B;
  złyznak = '?';
var x:integer; znak := char;
  t:array 0:63 of char;
  (* tablica translacji znaków *)
  z:array 0:7 of integer;
procedure zamień;
  (* liczbę x na znak *)
var s,c : integer;
begin s:=x div 32; c:=x mod 32;
  s:=z[s];
  if s < 0 then znak:=złyznak else
    if x >+ 16 then c:=9 end;
    znak:=t [16*x+c]
  end
end zamień;
begin
  z[0] :=0; z[1] :=3; z[2] :=2; z[3] :=1;
  z[4] :=1; z[5] :=-1; z[6] :=-1; z[7] :=-1;
  t[0] :=''; t[1] :='1'; t[2] :='2'; t[3] :='3';
  .....
  t[60] := '('; t[61] := '-'; t[62] := '"'; t[63] := ' ';

```

```

loop czytaj (x);
  if x = eci then
    repeat czytaj(x) until x = 0;
    pisz (ff);
  else while x = 0 do
    zamień; pisz (znak); czytaj (x)
  end;
  pisz (lf); piskonlinii
end
end
end strumień;
process zegar;
use tiktak; crsign;
begin
  loop wait (tiktak); send (crsign)
end
end zegar
process kontrolakolejności;
begin
  loop testkolejności; wait (tiktak)
end
end kontrolakolejności;
begin
  strumień, zegar, kontrolakolejności
end strumieńdanych.

```

4.6. Ocena języka

Język Moduła był wykorzystany do tworzenia oprogramowania systemowego [3, 36, 68], stąd też wynikają oceny jakości języka. Pozytywne cechy języka uwidaczniają się w trzech zakresach.

Podstawowe mechanizmy strukturalizacji programów, tzn. procesy oraz moduły okazały się łatwe i dogodne w użyciu. Doświadczenia praktyczne wskazują na proste przejście od specyfikacji funkcji systemu do jego struktury programowej oraz na możliwość projektowania programu metodą kolejnych uszczegółowień (metodą zstępującą [62]).

Bardzo atrakcyjna okazuje się koncepcja modułów sterowników urządzeń (device modula). Moduł ten, jako specjalny rodzaj modułu pośredniczącego, zapewnia całkowitą obsługę działającego urządzenia wejścia/wyjścia oraz dostarcza dogodny interfejs programowy z pozostałą częścią programu.

Ostatnią zaletą języka jest efektywność jego implementacji. Translator języka napisany w Pascalu ma jądro napisane w języku pewnej maszyny wirtualnej o wielkości około 200 słów. Także czas pracy tego jądra przy realizacji przekazywania informacji pomiędzy procesami jest krótki. O rozmiarze wymaganej pamięci i czasie pracy programu decydują zatem funkcje, jakie ma on realizować.

Niezależnie od zalet, Moduła ma także pewne słabości. Pierwsza wynika stąd, że język jest przeznaczony do realizacji małych, statycznych systemów, które całkowicie rezydują w pamięci operacyjnej. Ograniczenie to uniemożliwia zaprogramowanie pełnej klasy aktualnie spotykanych systemów operacyjnych.

Skojarzoną wadą jest brak pojęcia typu systemowego, co zmusza do oddzielnych deklaracji nawet identycznych modułów sterowników urządzeń.

Wreszcie ostatnia wynika z trudności, jakie pojawiają się przy weryfikacji programów w Moduli [3]. Wydaje się, że Moduła - podobnie jak Concurrent Pascal - odegrał już swą rolę, stwarzając podwaliny pod dalszy przyspieszony rozwój języków programowania współbieżnego. Niezależnie od tego, języki pozostaną w dalszym użyciu dla programowania prostych, specjalizowanych systemów operacyjnych. Wydaje się to uzasadnione popularnością minikomputerów PDP11, dla których zbudowano translatory języków.

ĆWICZENIA

1. Dwa procesory są połączone poprzez adaptory liniowe dwupiętrową linią komunikacyjną. Zaprojektować interfejs programowy pomiędzy programem a adapterami liniowymi. Zaprojektować protokół wymiany informacji pomiędzy dwoma procesorami (uwzględnić możliwość powstania przekłamań na linii komunikacyjnej). Zaprogramować w języku Moduła procesy komunikujące się przez linię komunikacyjną zgodnie z zaprojektowanymi zasadami współpracy z adapterami linii i protokołem wymiany informacji.

2. Problem telegramów (Henderson, Snowden [50]). Zadaniem programu jest przetwarzanie strumienia telegramów. Strumień ten stanowi ciąg liter, cyfr i odstępów, generowany poprzez adapter liniowy i jest zamieniany na porcje o określonej z góry długości w pewnym obszarze buforowym. Słowa w telegramie są oddzielane ciągami odstępów (spacji), a każdy telegram kończy się ciągiem ZZZZ. Strumień telegramów kończy telegram pusty. Każdy telegram musi być przetworzony w modelu obliczenia liczby słów podlegających taryfie oraz liczby słów, których długość przekracza $12 \times k$ znaków ($k=1,2,\dots$). Wynikiem działania programu ma być lista telegramów wraz z wymienionymi poprzednio informacjami służącymi do obliczenia opłaty za telegram. Zaprogramować opisany problem w języku Moduła, wykorzystać możliwe "równoległości" problemu.

3. Na bazie przykładu systemu rezerwacji biletów (przykład P1 w p.2.2) opracować pełną (użytkową) jego wersję, a następnie zaprogramować go w języku Moduła.

4. Przykład opisany i zaprogramowany w p. 4.5 rozbudować w taki sposób, aby operator za pośrednictwem terminala mógł sterować pracą czytnika kart i drukarki (startowanie i zatrzymywanie tych urządzeń oraz ustawianie tych urządzeń na pracę krokową, tzn. czytanie jednej karty i wydruk pojedynczej linii).

5. Napisać w języku Modula program symulujący działanie czytelników i pisarzy (ćw. 1, rozdz. 2).

6. Napisać w języku Modula program symulujący działanie filozofów (ćw. 5, rozdz. 2).

7. Napisać w języku Modula program przydzielania jednorodnych zasobów procesom, zgodnie z algorytmem bankiera (ćw. 7, rozdz. 3).

8. Napisać w języku Modula szkielet modułu pośredniczącego organizującego współpracę procesów wg zasad opisanych w zad. 8, rozdz. 3.

9. Przeanalizować możliwość wzajemnego zastępowania mechanizmów szeregowania krótko- i średnioterminowego w językach Modula i Concurrent Pascal.

5. PROCESY W ŚRODOWISKU ROZPROSZONYM

5.1. Wprowadzenie

Procesy w środowisku rozproszonym stanowią stosunkowo nowy kierunek w programowaniu. Proponowane podejścia w ramach kierunku wyrastają m.in. z prac Hoare'a [35], Brinch Hansena [12], Kahna [41,42], Liskov [48], Hewitta [29,30]. Część spośród tych prac, zwłaszcza [35,12], stworzyły podstawy do projektu języka Ada [40] (rozdział 6), który - można tego oczekiwać - odegra w zakresie programowania współbieżnego rolę podobną do Pascala w zakresie programowania sekwencyjnego.

Potrzeba rozwoju tego kierunku wynikła z dwóch zasadniczych przyczyn (rozdz. 1):

- rozwoju systemów wieloprocesorowych (skala mikro),
- rozwoju sieci komputerowych (skala makro).

Przyczyny te wskazują jednocześnie na wzrastające znaczenie komunikacji pomiędzy procesami. Przyjmowane zasady komunikacji stanowią znaczący wyróżnik propozycji różnych podejść. Zasady te można sklasyfikować, przedstawiając możliwe sposoby zachowania się procesu-nadawcy i procesu-odbiorcy. Dla wygody założymy, że procesy są połączone abstrakcyjnym kanałem, po którym mogą sobie przysyłać wiadomości.

Proces-nadawca może komunikować się z procesem-partnerem, zgodnie z następującymi, kolejno coraz bardziej złożonymi, wariantami:

N1. Proces-nadawca kompletuje wiadomości, a następnie bezwarunkowo wysyła je w odpowiedni kanał, po czym kontynuuje swe obliczenia.

N2. Proces-nadawca po skompletowaniu wiadomości oczekuje aż proces-odbiorca będzie w stanie wiadomości przyjąć.

N3. Proces-nadawca oczekuje na odbiór wiadomości przez proces-odbiorcę, po czym oczekuje nadal, aż na podstawie przesłanej wiadomości proces-odbiorca skompletuje i prześle w odpowiedzi wiadomość zwrotną.

Zachowanie się procesu-odbiorcy zasadniczo charakteryzuje sposób buforowania napływających do niego wiadomości. Można tu wyróżnić:

B1. Brak bufora wiadomości; wiadomość może być odebrana tylko wówczas, gdy jest oczekiwana.

B2. Istnieje bufor, ograniczony lub nieograniczony, pozwalający na przyjęcie w dowolnym momencie skończonej lub nieskończonej liczby wiadomości.

Wspólnie proces-nadawcę i proces-odbiorcę charakteryzuje sposób oczekiwania na zsynchronizowanie się partnera. Możliwe są tu dwie reakcje:

01. Nieograniczone oczekiwanie na partnera.

02. Oczekiwanie ograniczone z góry określonym odcinkiem czasu (okres przeterminowania - time-out), po którym, w razie braku zsynchronizowania, proces-nadawca przerywa oczekiwanie i powraca do kontynuacji swych obliczeń.

Kwestią związaną z komunikacją procesów jest sposób identyfikacji procesu-nadawcy przez proces-odbiorcę. Oczywiście proces-nadawca musi znać miejsce, do którego wiadomość chce przesłać, czyli identyfikator procesu-odbiorcy. Możliwe są tu sytuacje:

I1. Proces-nadawca jest dla procesu-odbiorcy procesem bezimiennym: proces-odbiorca na podstawie analizy wiadomości (analizy zapisanej w danym języku programowania) nie potrafi określić źródła jej powstania. (Pomija się tu możliwość takiej definicji wiadomości, w której procesy będą zobowiązane podawać swe identyfikatory.)

I2. Proces-odbiorca zawsze zna identyfikator procesu-nadawcy.

Pomijając wady i zalety różnych sposobów komunikacji, ze względu na potrzeby konkretnych aplikacji i możliwość implementacji, scharakteryzujemy sposoby komunikacji w omawianych dalej podejściach. Zasady komunikacji pomiędzy sekwencyjnymi procesami komunikującymi się Hoare'a charakteryzuje następujący zestaw omawianych cech:

(N2,B1,01,I2),

komunikację w procesach rozproszonych Brinch Hansena:

(N3,B2,01,I1),

komunikację w sieci procesów Kahna:

(N1,B2,01,I1),

komunikację pomiędzy zadaniami (procesami) w języku programowania Ada:

(N3,B1,01 lub 02,I1).

5.2. Komunikujące się procesy sekwencyjne Hoare'a

Pojęcie komunikujących się procesów sekwencyjnych oraz język programowania służący do ich opisu pochodzą od Hoare'a [35]. Program składa się z rodziny procesów **sekwencyjnych**, które mogą współpracować ze sobą wyłącznie poprzez bezpośrednie przesyłanie wiadomości pomiędzy parami procesów (brak wspólnie dostępnej pamięci). Przesyłanie wiadomości odbywa się przez wykonanie instrukcji wyjścia w procesie nadawczym oraz instrukcji wejścia w procesie odbiorczym.

Przedstawiony krótki opis tego podejścia prezentuje jedynie zasadnicze jego idee. Notacja jest inna od oryginalnej, bardziej zbliżona do notacji pascalskiej. Pojęcia typów zmiennych, deklaracji itd. nie będą tu rozważane.

Strukturę programów wyznacza instrukcja równoległego wykonania, postaci:

parbegin <id₁>::S₁||<id₂>...||<id_n>:S_n parend,

gdzie <id₁>, <id₂>, ..., <id_n> są identyfikatorami procesów w równoległych, zaś S₁, S₂, ..., S_n są ciągami instrukcji odpowiadającymi treści poszczególnych procesów. Procesy mogą być rodziną parametryzowaną indeksem, co znajduje odbicie w postaci

parbegin <id>(i:1..n)::ciąg instrukcji
z parametrem i>

parend

Przykłady instrukcji równoległego wykonania

parbegin
zachód:: ROZKLADANIE ||
x:: KOMPRESJA ||
wschód:: SKŁADANIE
parend

parbegin
pomieszczenie:: POKOJ ||
widelec (i:0..4):: WIDELEC ||
filozof (i:0..4):: FILOZOF
parend

Instrukcje równoległego wykonania mogą być zagnieżdżane. Przejście do realizacji nowej instrukcji wykonania równoległego można uważać za kreację nowych procesów, a jej zakończenie - za unicestwienie tych procesów. W tym sensie program stanowi dynamiczną rodzinę tych procesów.

Napisy złożone dużymi literami stanowią pewne, dalej zdefiniowane ciągi instrukcji.

Instrukcje wejścia i wyjścia są zdefiniowane przez napisy:

<identyfikator procesu>? <zmienna>
<identyfikator procesu>: <wyrażenie>

Komunikacja, jaką wyznaczają te instrukcje, odbywa się zgodnie z następującym schematem: Jeżeli w procesie P1 jest wykonywana instrukcja wejścia P2?v, to, aby została ona zrealizowana, w procesie P2 musi zostać wykonana skojarzona instrukcja wyjścia P1!e. Wykonanie pary instrukcji

wymaga więc synchronizacji czasowej ich realizacji. Gdy jedna z nich zostanie rozpoczęta wcześniej niż druga, następuje jej opóźnienie aż do momentu nastąpienia ich synchronizacji. Wynikiem wykonania pary tych instrukcji jest podstawienie wartości wyrażenia e uformowanego w procesie P2 pod zmienną v zadeklarowaną w procesie P1.

Instrukcje służące do wyrażania treści procesów są interesujące przede wszystkim z powodu znacznego stopnia niedeterminizmu, jaki dopuszczają. Instrukcjami tymi są instrukcja alternatywy i iteracji, stanowiące nieco rozbudowane wersje języka programowania Dijkstry. W wersji oryginalnej mają one postać:

$$\begin{array}{l} \text{if } B_1 \rightarrow S_1 \square \quad B_2 \rightarrow S_2 \square \quad \dots \square \quad B_n \rightarrow S_n \quad \underline{fi} \\ \text{do } B_1 \rightarrow S_1 \square \quad B_2 \rightarrow S_2 \square \quad \dots \square \quad B_n \rightarrow S_n \quad \underline{od} \end{array}$$

gdzie $B_1, B_2, \dots, B_n$ są wyrażeniami logicznymi, nazywane dozorami, $S_1, S_2, \dots, S_n$ ciągami instrukcji. W wersji rozbudowanej zamiast dozoru B_1 może wystąpić para $B_1; C_1$, gdzie C_1 jest instrukcją wejścia.

Sens instrukcji alternatywy jest następujący: Realizacja instrukcji rozpoczyna się od obliczenia wartości warunków logicznych $B_1, \dots, B_n$. Jeżeli żaden z nich nie jest prawdziwy, następuje zerwanie (zatrzymanie) obliczeń. Jeżeli pewne warunki są prawdziwe, to rozpoczyna się próba realizacji tych instrukcji wejścia C_1 , dla których odpowiadające warunki B_1 są prawdziwe. Zrealizowana zostanie tylko jedna spośród tych instrukcji wejścia, ta, dla której nastąpi najwcześniejsza synchronizacja ze skojarzoną instrukcją wyjścia. Jeżeli jest to instrukcja C_1 , to po jej zakończeniu realizuje się ciąg S_1 , po czym kończy się realizacja całej instrukcji alternatywy.

Realizacja instrukcji iteracji przebiega podobnie: rozpoczyna się więc od obliczeń wartości warunków logicznych. Gdy żaden z tych warunków nie jest prawdziwy, następuje zakończenie instrukcji (a nie zerwanie, jak przy poprzedniej instrukcji). Gdy pewne warunki są prawdziwe, rozpoczyna się próba wykonania odpowiednich instrukcji wyjścia, a po realizacji jednej z nich wybór odpowiedniego ciągu S_1 - tak jak w instrukcji alternatywy - po czym przedstawiony cykl obliczeń powtarza się, powracając do ponownego obliczania nowych wartości warunków logicznych.

Gdy żadnej z instrukcji wyjścia nie można wykonać z tego powodu, że odpowiednie procesy partnerskie zakończyły swe działanie, następuje także zakończenie instrukcji. (Jest to tzw. reguła rozproszonego stopu.) Jak okazuje się przy bliższych badaniach, ma ona uciążliwe konsekwencje w dowodzeniu własności programów i dlatego jest w różny sposób modyfikowana^{*)}.

^{*)} Np. G.A. LEVIN, D. GRIES, A proof technique for communicating sequential processes, Acta Informatica, vol. 15, 281-302, 1981.

Przykłady podane niżej pokazują zastosowanie omówionych instrukcji do konstrukcji treści procesów wymienionych w poprzednich przykładach. Pierwszy z nich reprezentuje program złożony z trzech procesów, których zadaniem jest kopiowanie ciągu znaków odczytanych z czytnika kart na drukarkę z zamianą dwóch kolejnych znaków ** na znak †.

ROZKŁADANIE =

```

var karta: array (1..80) of char;
  i:integer;
do czytnikkart? karta → i:=1;
  do i ≤ 80 → x! karta(i); i:=i+1 od;
  x! ' '
od;

```

KOMPRESJA =

```

var c : char;
do zachód?c →
  if c ≠ '*' → wschód! c []
  c = '*' → zachód? c;
  if c ≠ '*' → wschód! '*'; wschód! c []
  c = '*' → wschód! ' '
fi
fi
od;

```

SKŁADANIE =

```

var linia: array (1..125) of char;
  i:integer; c:char;
i:=1;
do x?c → linia(i):=c;
  if i ≤ 124 → i:=i+1 []
  i = 125 → drukarka! linia; i:=1
fi
od:
if i=1 → exit []
i>1 → do i ≤ 125 → linia(i):=' ';
      i:=i+1
od:
drukarka! linia
fi;

```

Komunikacja procesów z urządzeniami wejścia/wyjścia - jak widać z przykładu - odbywa się na takich samych zasadach, jak komunikacja pomiędzy procesami.

Instrukcja exit umieszczona wewnątrz instrukcji alternatywy oznacza tutaj jej zakończenie i przejście do wykonywania dalszych instrukcji.

Drugi przykład odnosi się do zadania Dijkstry o filozofach jedzącym spaghetti.

Pięciu filozofów spędza czas jedząc i myśląc. Dzielią oni wspólny pokój z okrągłym stołem otoczonym pięcioma krzesłami, z których każde należy do jednego z filozofów. Na środku stołu znajduje się misa spaghetti, a na jego obrzeżu pięć widelców - każdy pomiędzy dwoma krzesłami. Filozof, który poczuje głód wchodzi do pokoju, siada na swym krześle i ujmuje widelec leżący po jego lewej stronie. Ponieważ jedzenie spaghetti wymaga jednak dwóch widelców musi on także wziąć drugi widelec, leżący po jego prawej stronie. Po zjedzeniu filozof odkłada widelce i wychodzi z pokoju. Gdy jednak potrzebne widelce są zajęte przez sąsiadów filozofa musi on czekać na ich zwolnienie. Rozwiązanie tego problemu jest następujące:

FILOZOF =

do true →

MYŚLENIE;

pokój! wejście ();

widelec(i)!zajmij();

widelec((i+1)mod 5)!zajmij();

JEDZENIE;

widelec(i)!zwolnij();

widelec((i+1)mod 5)!zwolnij();

pokój!wyjście()

od;

Parametr *i* jest tutaj indeksem rodziny procesów FILOZOF. Występujące tu wyrażenia: wejście(), zajmij(), zwolnij() są wyrażeniami pustymi. Oznacza to, że realizacja instrukcji np.

pokój! wejście()

nie przekazuje żadnej informacji (w postaci jawnie określonych danych) od jednego z procesów rodziny FILOZOF do procesu pokój. Sens tej instrukcji polega na tym, że spełnia ona rolę synchronizatora sterowania pary procesów. Zatem wejście(), zajmij(), zwolnij() spełniają tutaj taką rolę, jaką odgrywają sygnały w języku Moduła (rozdz. 4).

WIDEEC =

do filozof(i)? zajmij() →

filozof(i)? zwolnij() □

filozof((i-1)mod 5)? zajmij() →

filozof((i-1)mod 5)? zwolnij()

od;

POKÓJ =

```

var liczbafil:integer;
liczbafil:=0;
do (i:0..4) filozof(i)? wejście( ) →
    liczbafil:=liczbafil+1 □
    (i:0..4)filozof(i)? wyjście( ) →
        liczbafil:=liczbafil-1
od:

```

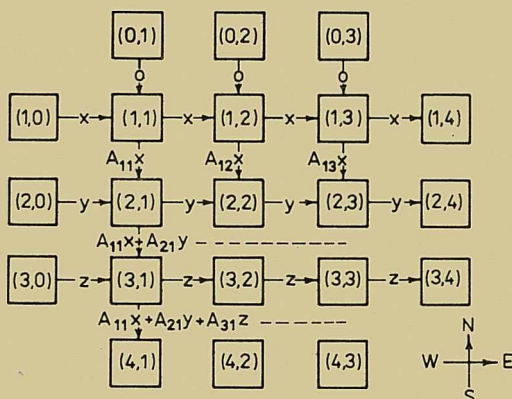
Zapis $(i:0..4)\text{filozof}(i)?\text{wejście}(\) \rightarrow \dots$ jest skrótem od zapisu:

```

filozof( )? wejście( ) → ... □
filozof(1)? wejście( ) → ... □
.....
filozof(4)? wejście( ) → ... .

```

Powyższe rozwiązanie dopuszcza sytuację, w której jednocześnie pięciu filozofów uchwyci lewy widelec. Prowadzi to do nieograniczonego czekania filozofów na prawy widelec. Sytuacji takiej można uniknąć kosztem dodatkowej komplikacji rozwiązania.



Rys. 5.1. Struktura przepływu danych pomiędzy procesami

Ostatni przykład odnosi się do równoległego mnożenia macierzy. Dana jest macierz kwadratowa A o wymiarach 3×3 . Trzy strumienie wejściowe reprezentują kolumny macierzy B . Trzy strumienie wyjściowe reprezentują kolumny produktu macierzy $B \times A$. Jeśli pominiemy początkowe opóźnienie, wyniki mają być generowane z tą samą czybkością, z jaką są podawane dane wejściowe. W konsekwencji jest wymagany wysoki stopień współbieżności. Rozwiązanie przyjmuje postać pokazaną na rys. 5.1.

Każdy wierzchołek centralny odbiera z zachodu (w) składową wektora B oraz sumy częściowe z północy (N). Wyprowadza natomiast składową wek-

tora wschód (E) oraz obliczoną sumę częściową na południe (S). Dane wejściowe są dostarczane przez wierzchołki zachodnie, wierzchołki północne reprezentują stałe zerowe, wierzchołki wschodnie są spływem danych wejściowych. Rozwiązanie ma postać:

parbegin

```
M(i+1..3,0)::ZACHÓD ||
M(0,j:1..3)::PÓŁNOC ||
M(1:1..3,4)::WSCHÓD ||
M(i:1..3,j=1..4)::CENTRUM
```

parend

gdzie

```
PÓŁNOC =
  do true → M(1,j):0 od;
WSCHÓD =
  var x:real;
  do M(i,3)?x → exit od;
CENTRUM =
  var x,s:real;
  A:array(i:1..3,j:1..3) of real;
  do M(1,j-1)?x →
    M(j,j+1):x;
    M(i-1,j)?s;
    M(i+1,j) ! A(i,j)*x+s
  od;
```

ZACHÓD oraz POŁUDNIE są niedefiniowanymi tutaj rodzinami procesów użytkowników.

5.3. Procesy rozproszone Brinch Hansena

Podejście Brinch Hansena [11] wprowadzające pojęcie procesów rozproszonych jest zbliżone do przedstawionego w poprzednim punkcie podejścia Hoare'a. Zasadnicze różnice dotyczą sposobu komunikowania się procesów. Wprowadzony język stanowi modyfikację języka Concurrent Pascal; zasadniczym elementem jest tu eliminacja pojęcia monitora jako jednostki programowej związanej z koncepcją procesów działających we wspólnym środowisku.

Podstawowym komponentem programu jest proces o następującej budowie:

```
process <identyfikator>;
  <zmienna lokalna>;
  <procedury ogólnodostępne>;
```


<instrukcja inicjalizująca>

end;

Program składa się ze stałej liczby procesów. Procedury ogólnodostępne są procedurami, które mogą być wywoływane zarówno przez proces, w którym są zdefiniowane, jak również przez inne procesy - wywołania zewnętrzne. Proces wykonuje dwa rodzaje działań: obliczenia instrukcji inicjalizującej oraz obliczania procedur ogólnodostępnych dla wywołań zewnętrznych. Czynności te są wykonywane na przemian: proces rozpoczyna obliczenia od realizacji instrukcji inicjalizującej; czynność ta jest kontynuowana aż do zakończenia procesu lub napotkania na warunek, który może zostać spełniony na skutek działalności innych procesów; dalsze działanie proces kontynuuje na skutek wywołań swych procedur ogólnodostępnych przez inne procesy, następnie albo oczekuje na dalsze akcje zewnętrzne, albo powraca do obliczenia instrukcji inicjalizującej; po zakończeniu instrukcji inicjalizującej proces istnieje nadal i obsługuje wywołania zewnętrzne. Przeplot wykonywanych operacji nie jest zatem dokonywany poprzez podział czasu, lecz jest programowany przez proces.

Procedury ogólnodostępne definiują swe parametry wejściowe (in) oraz wyjściowe (out), zmienne lokalne i działający na nich ciąg instrukcji:

```
procedure <identyfikator> (in <param we>, out <param wy>);
  <zmiennie lokalne>;
  <ciąg instrukcji>
end;
```

Wywołanie procedury ogólnodostępnej R zdefiniowanej w procesie Q następuje w wyniku wykroczenia instrukcji

```
call Q.R (<wyrażenia>, <zmiennie>)
```

Przed wykonaniem procedury R wartości <wyrażen> zostają podstawiane pod <param we>, a po jej zakończeniu odpowiednie wartości <param wy> zostają podstawione pod <zmiennie>.

Podobnie jak w wypadku komunikujących się procesów sekwencyjnych, możliwa jest definicja indeksowanej rodziny procesów. Dla uproszczenia przyjmujemy, że ma ona taką samą postać, jak w p. 5.2.

Pozostałe elementy obsługi wywołania procedury (wybór jednego spośród wielu żądań obsługi, opóźnianie innych żądań) są takie same, jak przywołaniu procedur wejściowych monitora w Concurrent Pascalu (rozdz. 3).

Niedeterminizm jest wprowadzony do języka przez dwie instrukcje dozorowane: alternatywy i iteracji - takie same, jak u Hoare'a w poprzed-

nim punkcie 5.2 (wersja oryginalna Dijkstry) - oraz dwie instrukcje dozorowanego regionu:

```
when  $B_1 \rightarrow S_1$   $\square$  ...  $\square$   $E_n \rightarrow S_n$  end;  
cycle  $B_1 \rightarrow S_1$   $\square$  ...  $\square$   $E_n \rightarrow S_n$  end;
```

gdzie $B_1, \dots, B_n$ oraz $S_1, \dots, S_n$ oznaczają - jak poprzednio - warunki logiczne i ciągi instrukcji.

Wykonanie instrukcji when polega na oczekiwaniu do momentu aż jeden z warunków $B_1, \dots, B_n$ przyjmie wartość true, a następnie na wykonaniu odpowiadającego mu ciągu instrukcji. Instrukcja cycle stanowi nieograniczone powtarzanie w pętli instrukcji when.

Zastosowanie omówionych konstrukcji przedstawiają podane niżej przykłady. Pierwszym przykładem jest proces modelujący działanie semafora całkowitoliczbowego, inicjalizowanego z wartością zero; z operacjami czekaj i kontynuuj:

```
process semafor;  
  var s:integer;  
  procedure czekaj;  
    when  $s > 0 \rightarrow s := s - 1$  end;  
  end czekaj;  
  procedure kontynuuj;  
     $s := s + 1$   
  end kontynuuj;  
   $s := 0$   
end semafor;
```

Instrukcja inicjalizacji przypisuje wartość zero do zmiennej s, po czym zostaje zakończona. Proces jednak istnieje i jest gotowy do obsługi wywołań zewnętrznych

```
call semafor, czekaj  
call semafor, kontynuuj.
```

Drugi przykład jest uproszczoną wersją rozwiązania problemu o jedzących filozofach. Przedstawiamy je poniżej bez komentarza.

```
process filozof(i:0..4);  
  do true  $\rightarrow$  MYŚLENIE;  
    call stół.zajmij(i);  
    JEDZENIE;  
    call stół.zwolnij(i)  
  od;  
end filozof;
```



```

process stół;
var zajęte: set of 0..4;
procedure zajmij(in i:integer);
  when ([i,(i+1)mod 5] and zajęte)=[ ] →
    zajęte:=zajęte+[i,(i+1)mod 5]
  end
end zajmij;

procedure zwolnij(in i:integer);
  zajęte:= zajęte-[i,(i+1)mod 5];
end zwolnij;
zajęte:=[ ]
end stół;

```

Ostatni, trzeci przykład wprowadza rodzinę procesów sortujących m liczb. Dane są wprowadzane przez proces sortuj(1), który zapamiętuje najmniejszą liczbę, a pozostałe przekazuje do procesu sortuj(2). Proces sortuj(2) znów zapamiętuje najmniejszą spośród przekazanych mu liczb, a pozostałe przekazuje do procesu sortuj(3) itd. Gdy zostanie zakończone wprowadzanie m liczb, zostaną one ułożone we wzrastającej kolejności w pamięci kolejnych procesów, po czym mogą być przesłane z powrotem na miejsce, z którego wyszły, tj. z procesu użytkownika zdefiniowanego następująco:

```

process użytkownik;
  var A: array (1..m) of integer;
  i : integer;
  for i:=1 to m do
    call sortuj(1).przełącz(A(i));
  for i:=1 to m do
    call sortuj(1).odbierz(A(i))
  end użytkownik;

```

Rodzina procesów sortuj zawiera n procesów, przy czym zakłada się, że $n \geq m$. W definicji procesu występuje struktura danych typu ciąg (sequence), z operacjami (procedurami) standardowymi pobrania ostatniego elementu tego ciągu (get), dopisania do końca ciągu (put) oraz z funkcją standardową określającą aktualną długość ciągu (length).

```

process sortuj(i:1..n);
var stos: sequence 2 of integer;
  p,t:integer;
procedure przełącz (in c:integer);
  when stos.length 2 stos.put(c) end;
end przełącz;

```

```

procedure odbierz (out v:integer);
    when stos.length=1  $\rightarrow$  stos.get(v) end;
end odbierz;
stos:=[ ];
p:=0;
cycle stos.length=2  $\rightarrow$ 
    if stos[1]  $\leq$  stos[2]  $\rightarrow$  t:=stos[2];
        stos:= stos[1] []
        stos[1]  $\geq$  stos[2]  $\rightarrow$  r:= stos[1];
        stos := stos[2]
    fi:
    call sortuj(i+1).przekaz(t);
    p:=p+1 []
    (stos.length=0) and (p > 0)  $\rightarrow$ 
    call sortuj(i+1).odbierz(t)
    p:=p-1
end
end sortuj;

```

Proces sortuj jest w równowadze, gdy zapamiętuje tylko jedną liczbę. W wypadku braku tak zrozumianej równowagi proces podejmuje jedną z akcji:

- jeżeli proces pamięta dwie liczby, mniejszą z nich zachowuje, a większą przekazuje do procesu sąsiedniego,
- jeżeli proces nie pamięta żadnej liczby, zwraca się o pewną liczbę do procesu sąsiedniego.

5.4. Sieci procesów Kahna

Omawiane niżej podejście Kahna [41,42] charakteryzują, w stosunku do podejść poprzednich, inne zasady komunikacji pomiędzy procesami oraz dynamiczna struktura procesów (szerokie możliwości kreacji i likwidacji procesów). Język programowania definiowany przez Kahna bazuje na pojęciu procesu - rozumianego jak poprzednio - oraz na pojęciu kanałów komunikacyjnych, przez które procesy mogą wymieniać wiadomości. Kanały przenoszą informacje tylko w jednym kierunku od procesu-nadawcy do procesu-odbiorcy; mają one bufora nieograniczonej długości, w których są magazynowane napływające wiadomości (kolejka FIFO). Procesy wraz z łączącymi je kanałami tworzą tzw. sieć procesów.

Deklaracja procesów ma postać:

```

process <identyfikator>(<lista parametrów>;
    <ciało procesu>;
endprocess

```


<Lista parametrów> składa się ze zwykłych parametrów, służących do ustalania wartości początkowych lokalnych zmiennych procesu podczas jego inicjalizacji oraz tzw. portów. Te ostatnie służą do zdefiniowania kanałów łączących dany proces z innymi procesami. Na przykład napis

```
process pośrednik (x:integer;  
    QI: in integer; QO out integer);
```

definiuje nagłówek procesu pośrednik, który ma jeden zwykły parametr formalny *x* typu integer, komunikowany przez wartość oraz dwa parametry portowe; QI oznaczający identyfikator pewnego kanału komunikacyjnego (kanału wchodzącego do procesu pośrednik), po którym można przesyłać informacje typu integer; QO oznaczający początek pewnego kanału komunikacyjnego (kanału wychodzącego od procesu pośrednik), po którym można przesyłać informacje także typu integer.

<Ciało procesu> definiuje zmienne lokalne oraz ciąg instrukcji, który zawiera zwykłe instrukcje działań na zmiennych lokalnych oraz instrukcje wejścia/wyjścia (komunikacji z innymi procesami) i instrukcje rekonfiguracji sieci procesów.

Komunikację procesy prowadzą przez instrukcje wejścia/wyjścia stanowiące odwołanie się do łączących je kanałów komunikacyjnych. Instrukcja odbioru (wejścia) z portu wejściowego *A* ma postać funkcji standardowej *get(A)*. Funkcja ta przyjmuje wartość pierwszego elementu w buforze kanału. Jeśli bufor jest pusty, to wykonanie *get(A)* opóźnia się tak długo, aż jakiś proces umieści w nim jakąś wartość. Po wykonaniu funkcji *get* pobrany element zostaje z bufora usunięty. Instrukcja nadawania (wyjścia) do wyjściowego portu *B* jest procedurą standardową o postaci *put (<wyrażenie>,B)*. Procedura *put* obliczoną wartość <wyrażenia> umieszcza w kanale o porcie *B*. Procedury *get*, *put* są jedynymi operacjami, jakie można wykonywać z udziałem kanałów komunikacyjnych.

Dopuszcza się sytuację, gdy liczba procesów-odbiorców dzieli ten sam kanał wejściowy. W tym wypadku wszystkie procesy-odbiorniki otrzymują na swym wejściu takie same ciągi danych.

Zwraca uwagę brak możliwości testowania stanu zajętości kanału. Nieuwzględnienie tej możliwości jest świadomie podjętą decyzją uzasadnioną tym, że upraszcza to znacznie semantykę języka, gdyż wyklucza się czasowo zależne zachowanie się kanałów. Warto zauważyć, że gdyby w procesach istniała możliwość badania zajętości kanału, wówczas rezultat obliczeń programu mógłby być różny w zależności od strategii przydziału czasu procesora. Strategii tych może być wiele: od sytuacji, gdy istnieje po jedynym procesorze z podziałem czasu do sytuacji, gdy każdy proces dysponuje własnym procesorem. Foruszone tu kwestie pozostają jednak poza zakresem rozważań skryptu.

Sieć procesów można uważać za pewien graf skierowany, którego wierzchołki reprezentują procesy, a łuki - kanały komunikacyjne. Podczas obliczeń programu graf ten można zmieniać metodą kolejnych uszczegółowień. Oznacza to, że wierzchołki grafu mogą być zastępowane przez nowe grafy. Jedyne warunki, jakie musi być przy tym zachowany, wymaga, aby łuki wejściowe i wyjściowe nowo powstałego grafu kojarzyły się z łukami wejściowymi i wyjściowymi zastępowanego wierzchołka. Instrukcja rekonfiguracji ma postać:

doco <treść> closeco

(doco od doconcurrently), gdzie treść składa się z dwóch części:

- deklaracji kanałów komunikacyjnych (channels) w nowo tworzonej grafie;
- listy wywołań (inicjalizacji) procesów; parametry portowe w tych wywołaniach muszą być związane albo z nowo zdefiniowanymi kanałami, albo z kanałami procesu, w którym jest wykonana dana instrukcja rekonfiguracji; zainicjowane procesy rozpoczynają swe obliczenia równolegle do procesu, który je uaktywnił.

Przykład prostej sieci procesów z dynamicznym tworzeniem ich struktury jest podany poniżej:

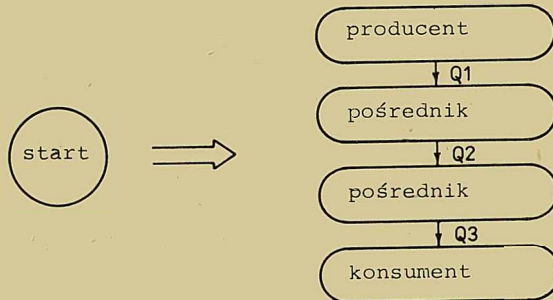
```
(* deklaracja procesów *)
process producent (out Q0:integer);
  var n:integer;
  n:=0;
  cycle n:=n+1;put(n,Q0) end;
endprocess;
process pośrednik (a:integer;
  Q0: in integer; Q0: out integer);
  cycle put(a + get(QI),Q0) end;
endprocess;
process konsument (QI: in integer);
  var i:integer;
  for i:=1 to 20 do print (get(QI));
endprocess;
process start;
  doco channels Q1,Q2,Q3:integer
    producent (Q1);
    pośrednik (1,Q1,Q2)
    pośrednik (-1,Q2,Q3);
    konsument (Q3)

  closeco
endprocess;
```



```
( * treść programu * )
begin
  doco start closeco
end
```

Początkowa struktura sieci, złożona tylko z jednego procesu start, prze-
radza się w sieć złożoną z czterech procesów (rys. 5.2).



Rys. 5.2. Zmiana konfiguracji sieci procesów

Wynikiem działania programu będzie ciąg liczb 1,2,...,20 wydrukowa-
ny przez proces konsument.

Ciekawszy jest przykład programu obliczającego liczby pierwsze (si-
to Eratosthenesa). Dla każdej znalezionej liczby pierwszej jest kreowa-
ny proces nowy filtr przez proces sito. Zadaniem procesu filtr jest wy-
szukanie wszystkich wielokrotności znalezionej liczby pierwszej.

```
( * deklaracja procesów * )
process generator (Q0:out integer);
  var n: integer;
  n:=1;
  cycle n:=n+1; put(n,Q0) end
endprocess;
process filtr (pierwsza: integer,
  QI: in integer; QO: out integer);
  var n:integer;
  cycle n:= get(QI);
    if (n mod pierwsza)  $\neq$  0 then
      put(n,QO)
    end
endprocess;
process sito (QI: in integer; QO: out integer);
  var pierwsza: integer;
```

```

    pierwsza := get(QI);
    put (pierwsza, QO);
    doco channels Q: integer;
        filtr (pierwsza QI,Q);
        sito (Q,QO)

    closeco;
endprocess;
process wyjście (QI: in integer);
    cycle print (get (QI)) end;
endprocess;
(* treść programu *)
begin
    doco channels Q1,Q2: integer;
        generator (Q1);
        sito (Q1,Q2);
        wyjście (Q2)

    closeco
end.

```

Przedstawione konstrukcje są wystarczające do programowania dowolnych algorytmów równoległych, czasowo niezależnych. Wprowadzoną notację możemy nieco ulepszyć, uzyskując przy tym większą przejrzystość i elegancję programów. Większość procesów ma jedno wyjście, w tym wypadku można procesy traktować jako funkcje i używać je do konstrukcji wyrażeń. W wywołaniach procesów w instrukcji rekonfiguracji wyrażenia takie mogą być dostarczane tam, gdzie oczekuje się kanałów wyjściowych. Przykładowo poprzedni program można zastąpić formą:

```

process generator( ) QO: integer
endprocess;
process filtr (pierwsza: integer;
    QI: in integer) QO: integer;
.....
endprocess;
process sito (QI: in integer) QO: integer;
.....
endprocess;
process wyjście (QI: in integer);
.....
endprocess;
begin
    doco wyjście (sito(generator( )))
    closedo
end.

```


Nowa, funkcjonalna notacja jest wygodna także z tego powodu, że wiele kanałów może być tworzonych w sposób bezpośrednio niewidoczny. Notacja wymaga jednak zmiany instrukcji rekonfiguracji.

Pierwsza zmiana polega na możliwości przekazywania strumieni informacji z jednego kanału wejściowego na drugi kanał wyjściowy - wyjaśnia to przykład rekonfiguracji w procesie sito:

```
doco sito(filtr(pierwsza,QI))  $\Rightarrow$  Q0
closeco
```

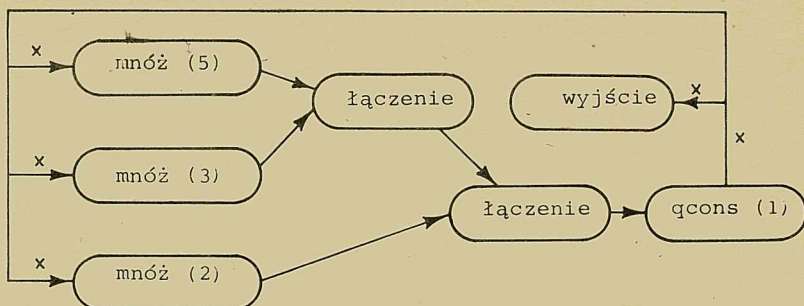
Innym przykładem odnoszącym się do tej sytuacji jest proces

```
process qcons(a:integer, QI: in integer) Q0: integer;
  put(a:Q0);
  doco QI  $\Rightarrow$  Q0 closeco
endprocess,
```

który wysyła liczbę a w kanał wyjściowy, potem spina kanał wejściowy z wyjściowym, a następnie znika.

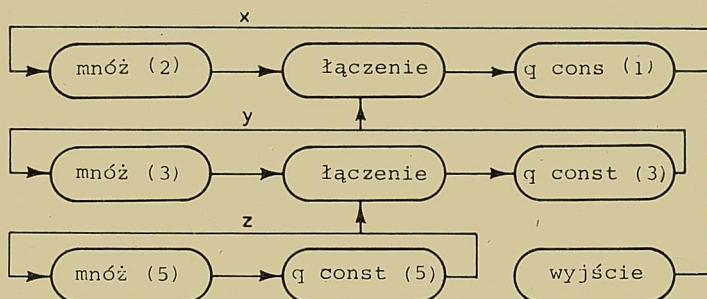
Drugą zmianą w instrukcji rekonfiguracji wyjaśnia kolejny przykład. Jest on związany z problemem generacji pierwszych n liczb postaci $2^a 3^b 5^c$ ($a, b, c > 0$) we wzrastającej kolejności bez powtórzeń lub braków:

```
(* deklaracja procesów *)
process łączenie (QI1,QI2: in integer) Q0: integer;
  var i1,i2: integer;
  i1:= get (QI1); i2:=get(QI2);
  loop i1<i2 then put(i1,Q0); i1:=get(QI1)
  elseif i1>i2 then put(i2,Q0); i2:=get(QI2);
  else put(i1,Q0); i1:=get(QI1); i2:=get(QI2);
  endloop;
endprocess;
process qcons (a:integer,QI: in integer)Q0: integer;
  ..... treść jak wyżej .....
endprocess;
process wyjście (QI: in integer)
  ..... treść jak w poprzednim programie .....
endprocess
(* treść programu *)
begin
  doco wyjście(x) where channels x is
    qcons(s,łączenie(mnóż(2,x),łączenie(mnóż(3,x),
      mnóż(5,x))))
  closedo
end.
```



Rys. 5.3. Struktura sieci procesów (I)

Ostatnia instrukcja rekonfiguracji wyznacza następującą strukturę sieci procesów (rys. 5.3). Struktura ta jednocześnie tłumaczy zastosowany algorytm obliczeń.



Rys. 5.4. Struktura sieci procesów (II)

Inną bardziej efektywną, ze względu na eliminację redundancji generowanych liczb, jest struktura pokazana na rys. 5.4, która wynika z następujących instrukcji rekonfiguracji

```

begin
  doco wyjście(x) where
    channels x,y,z are
      qcons (1,łączenie(mnóż(2,x),y)),
      qcons (3,łączenie(mnóż(3,y),z)),
      qcons (5,mnóż(5,z))
    closeco
end.

```


ĆWICZENIA

1. Rozwiązanie problemu jedzących filozofów przedstawione w p. 5.2 zmodyfikować w taki sposób, aby żadnemu z filozofów nie groziła możliwość wygłodzenia z powodu sytuacji nieskończonego oczekiwania na widelce.
2. Algorytm sortowania liczb przedstawiony w p. 5.3 przetransformować na program w języku Hoare'a.
3. Problem czytelników i pisarzy (ćw. 1, rozdz. 2) rozwiązać w konwencji rozproszonych procesów Brinch Hansena.
4. Zaprogramować w języku Brinch Hansena proces realizujący opóźnianie innych procesów o zadany odcinek czasu. Założyć, że dana jest procedura realizująca opóźnienie jednostkowe.
5. Zanalizować przykład programu generacji liczb pierwszych z p. 5.4. Narysować strukturę sieci procesów i określić kolejne postaci jego rekonfiguracji.
6. Zaprogramować w języku Kahna algorytm sortowania liczb (np. z p. 5.3).
7. Zaprojektować sieć procesów Kahna realizujących mnożenie macierzy zgodnie z algorytmem z p. 5.2.

6. MECHANIZMY RÓWNOLEGŁOŚCI W JĘZYKU PROGRAMOWANIA ADA

6.1. Wprowadzenie

Język programowania Ada ma bardzo ciekawą historię. Pomysł zdefiniowania i wprowadzenia do powszechnej praktyki języka wysokiego poziomu przeznaczonego zarówno do oprogramowania użytkowego, jak i systemowego powstał w latach 1973-74. Głównym inicjatorem i koordynatorem prac była międzynarodowa grupa robocza, powołana i finansowana przez amerykańskie ministerstwo obrony. Prace nad nowym językiem były prowadzone w formie konkursu, do którego początkowo przystąpiło ponad sto zespołów z Europy i USA. Prace przebiegały etapowo: na podstawie początkowo sformułowanych wymagań i postulatów powstawały pierwsze szkice projektów. Były one poddawane weryfikacji i selekcji, a na ich podstawie formułowano kolejne, bardziej precyzyjne wymagania. Powstał w ten sposób ciąg oficjalnych dokumentów z wymaganiami, określonych oryginalnymi nazwami: Strawman,..., Woodman,..., Ironman,..., Steelman. Ostatecznie w 1979 r. przez ostatnie sito konkursu przeszedł język Ada (nazwany tak przez twórców na cześć Augusty Ady Byron, 1815-1852, uznanej przez historyków informatyki za pierwszą programistkę; programowała ona maszyny Babbage'a). Język ten, godząc wiele sprzecznych wymagań [70], np. wysoki poziom abstrakcji z łatwością implementacji i efektywnością kodu wynikowego, stanowi obiekt dalszych udoskonaleń, a także krytyk [58]. Obecnie istnieje już kilka implementacji języka Ada.

W bieżącym rozdziale omawia się tylko mechanizmy programowania równoległego w języku Ada. Sposób ich prezentacji jest zupełnie niesformalizowany. Ze względu na przejrzystą notację przypominającą w dużym zakresie notację pascalską, podane opisy i przykłady nie powinny powodować trudności zrozumienia, mimo braku pełnego opisu języka.

6.2. Struktura programów

Podstawową jednostką strukturalną w języku Ada jest zadanie (task). Jest to jednostka syntaktyczna służąca definicji procesu. Inaczej niż w omówionych poprzednio językach programowania, zadania w języku Ada mogą być zagnieżdżane. Zadanie składa się z dwóch części: specyfikacji pośrednictwa (interfejsu) zadania z jego otoczeniem oraz z opisu działania zadania (ciało zadania).

Specyfikacja pośrednictwa zadania zawiera nagłówek identyfikujący zadanie oraz definicję obiektów (atrybutów) zadania dostępnych w jego otoczeniu. Są to m.in. stałe, typy, zastrzeżenia (exception) i porty (entry). Przykładem specyfikacji jest:

```
task bufor is
  rozmiarpakietu: const integer: 256;
  type pakiet is array (1..rozmiarpakietu) of char;
  entry czytaj (v: out pakiet);
  entry pisz (e: in pakiet)
end bufor;
```

Wymienione: stała, typ oraz porty (w terminologii Concurrent Pascala odpowiedniki procedur wejściowych) czytaj, pisz stanowią atrybuty dostępne na zewnątrz zadania bufor. Interesujące jest wyróżnienie wejściowych (in) i wyjściowych (out) parametrów procedur.

Ciało zadania składa się z deklaracji obiektów lokalnych (struktury danych) oraz z ciągu instrukcji implementujących porty z części specyfikującej

```
task body bufor is
  rozmiarbuf: const integer := 10;
  buf: array (1..rozmiarbuf) of pakiet;
  do, z: integer range 1..rozmiar_buf := 1;
  licznik: integer range 1..rozmiar_buf := 0;
begin
  ... ciąg instrukcji opisujących porty czytaj,
  pisz będzie opisany dalej ...
end bufor;
```

Aby rozpocząć obliczenia zadania, należy użyć instrukcji inicjalizującej (initiate). Przykładowo instrukcja

```
initiate bufor, producent, konsument,
```

wykonana w pewnym zadaniu, nazywanym inicjalizatorem, powoduje rozpoczęcie obliczeń zadań bufor, producent oraz konsument. Ich obliczenia będą wykonywane równolegle z zadaniem - inicjalizatorem. Każde zadanie (poza głównym, programem) musi być inicjalizowane. Zadaniem nadrzędnym w stosunku do danego zadania nazywa się zadanie, które zawiera jego deklarację. Należy podkreślić, że zadanie nadrzędne może być inne od inicjalizatora, jakkolwiek oba muszą mieć dostęp do zainicjalizowanego zadania. Rozpatrzmy przykład

```
task body T1 is
  task T2 is
```

```

.....
end T2;
task body T8 is
.....
end T2;
task T3 is
.....
end T3;
task body T3 is
.....
    initiate T2;
    .....
    end T2;
begin
.....
    initiate T3;
    .....
end T1;

```

Zadanie T1 jest nadrzędne względem T2, ale T2 jest inicjalizowane przez T3, a nie przez T1.

Normalne zakończenie zadania polega na osiągnięciu przez sterowanie end na końcu ciała zadania. Jeżeli zadanie, które ma się zakończyć, jest nadrzędne w stosunku do pewnych innych zadań, to jest ono opóźniane aż do momentu zakończenia wszystkich jego zadań podrzędnych. Zadanie może także kończyć się w wyniku realizacji instrukcji zerwania (abort). Przykładowo, instrukcja

```
abort T1,T2;
```

powoduje bezwarunkowe zakończenie zadań T1,T2 oraz wszystkich ich zadań podrzędnych. Inne zadania, które usiłują skomunikować się z zadaniami zakończonymi, zostaną przed tym powstrzymane przez powstanie zastrzeżenia standardowego "błąd zadania". (Zastrzeżenia, sposób ich deklaracji, powstawania i reagowania na nie stanowią element systemu zarządzania błędami, który jest wbudowany w definicję języka. System ten nie jest tutaj omawiany.)

Każdemu z zadań można przypisać stopień pilności - priorytet służący algorytmom przydziału czasu procesora do wyboru odpowiedniej strategii obsługi zadania. Priorytet raz przypisany zadaniu można zmieniać dynamicznie przez wywołanie procedury modyfikacji priorytetu (set priority).

W razie potrzeby utworzenia wielu podobnych zadań możliwe jest indeksowanie zadań - tworzenie rodziny zadań (task families). Przykładem

może być użycie wielu kopii zadań obsługi urządzeń wejścia/wyjścia.
 Fragment programu

```
task sterownikdalekopisu (1..100) is
  type linia is array (1..32) of char;
  entry piszlinię (tekst: in linia);
  entry czytajlinię (tekst: out linia);
end sterownikdalekopisu;
task body sterownikdalekopisu is
  .....
end sterownikdalekopisu;
```

deklaruje 100 kopii zadań, z których każde jest identyfikowane przez odpowiedni wskaźnik, np. instrukcja

```
initiate sterownikdalekopisu(3);
```

spowoduje aktywację trzeciej kopii zadania.

W języku istnieje możliwość wyboru, na etapie kompilacji programu, sposobu przydziału pamięci procesom. Polecenia

```
pragma creation (static)
```

lub

```
pragma creation (dynamic)
```

umieszczone w opisie zadania dla tłumacza powodują wybór statycznego lub dynamicznego przydziału pamięci. Możliwość dynamicznego przydziału pamięci jest szczególnie ważna wtedy, gdy program składa się z dużej liczby zadań, przy czym tylko część z nich jest jednocześnie zainicjowana.

Poza omówionymi sposobami statycznej deklaracji zadania lub rodziny zadań istnieje możliwość dynamicznego tworzenia nowych egzemplarzy podczas wykonywania programu. Zadanie, które może być dynamicznie powielane, jest wyróżnione przez dodanie słowa generic do części specyfikującej zadania, np.

```
generic task bufor is
  rozmiarpakietu: const integer: 256;
  .....
  entry pisz (a: in palmet)
end bufor;
```

Jeżeli w programie zachodzi konieczność dynamicznego utworzenia egzemplarza zadania bufor, można to uczynić przez użycie konstrukcji:

```
task nowybufor is new bufor
```

a odwołania do utworzonego zadania mogą mieć postać

```
nowybufor.pisz (daneprod);
nowybufor.czytaj(danekons);
```

6.3. Komunikacja pomiędzy zadaniami

Komunikacja pomiędzy zadaniami w języku Ada stanowi mieszaną formę komunikacji pomiędzy sekwencyjnymi procesami komunikującymi się (SPK) Hoare'a oraz procesami rozproszonymi (PR) Brinch Hansena omówionymi w rozdziale 5. W SPK komunikacja zachodzi przez skojarzenie instrukcji wejścia i instrukcji wyjścia, w których procesy wskazują wzajemnie na siebie. W PR komunikacja zachodzi w rezultacie wywołania w jednym procesie procedury wejściowej drugiego procesu. Komunikacja w języku Ada swą postacią przypomina komunikację w PR, tzn. jedno zadanie wywołuje port (procedurę wejściową) drugiego zadania z listą parametrów aktualnych zapewniających dwukierunkowy przepływ danych. Natomiast - jak zobaczymy - sposób, w jaki wywołane zadanie tworzy odpowiedź jest bliższy SPK niż PR.

Bezpośrednią różnicą jest to, że komunikacja w SPK jest symetryczna w tym sensie, że oba komunikujące się procesy muszą wzajemnie się wywoływać, podczas gdy w Adzie oraz w PR komunikacja jest asymetryczna w tym sensie, że wywołane zadanie nie musi wywoływać, ani też nawet nie zna, zadania które je wywołuje.

Komunikacja pomiędzy zadaniami w Adzie odbywa się dzięki wykorzystaniu instrukcji wołania portów (w zadaniu wołającym) oraz instrukcji akceptacji (w zadaniu wołanym). Przykładowo, odwołując się do przykładów z poprzedniego punktu, wywołaniami portów mogą być

```
bufor.pisz (daneproducenta),
bufor.czytaj (danekonsumenta).
```

Wywoływane porty muszą być oczywiście zadeklarowane w zadaniu bufor. Na podstawie tej deklaracji sprawdza się poprawność parametrów aktualnych.

Instrukcja akceptacji (accept) spełnia rolę podobną do treści procedury. Dokładniej, służy ona do wskazania instrukcji, które mają być wykonane w wyniku wywołania konkretnego portu. Przykładowo instrukcje akceptujące wywoływanych wyżej portów mają postać:

```
accept pisz (e: in pakiet) do
    buf (dox) := e;
end pisz;
accept czytaj (v: out pakiet) do
    v := buf(zx)
end:
```


W instrukcji akceptacji jest powtórzony nagłówek portu występujący w części specyfikującej zadania w celu przypomnienia zakresu jego parametrów. Zmienne *dox*, *zx* są liczbami całkowitymi wskazującymi odpowiednio koniec i początek bufora, i są zadeklarowane w ciele zadania (kompletny przykład jest pokazany dalej). Należy podkreślić, że zmienne te nie mogą być zmieniane wewnątrz instrukcji akceptacji. Ponieważ instrukcje akceptacji są wykonywane w rozłącznych odcinkach czasu, jest wskazane, aby instrukcje te były możliwie krótkie i nie zawierały takich działań, które mogłyby być wykonane poza ich wnętrzem.

Synchronizacja pomiędzy komunikującymi się zadaniami opiera się na protokole "rendez vous" Hoare'a (p. 5.2). Zgodnie z tym protokołem należy rozważyć dwie możliwe realizacje umówionego spotkania (*rendez vous*) zależne od tego, czy zadanie wywołujące port innego zadania dokona tego przed, czy po tym momencie, w którym rozpocznie się realizacja odpowiedniej instrukcji akceptacji. W obu wypadkach zadanie, które pierwsze osiągnie umówiony punkt spotkania jest opóźniane aż do chwili, gdy punkt ten osiągnie także drugie zadanie. Gdy to nastąpi, wówczas parametry wejściowe (*in*) zostaną przekazane do portu. Następnie, po wykonaniu instrukcji akceptacji, parametry wyjściowe (*out*) zostaną przekazane z powrotem do wołającego zadania. Od tego momentu oba zadania wykonują się dalej niezależnie od siebie. Z każdą procedurą wejściową jest związana kolejka, w której przechowuje się wszystkie wołania kierowane do danego portu. Wołania z kolejki są obsługiwane w kolejności zgłoszeń.

6.4. Niedeterminizm

Sekwencyjne struktury sterowania w Adzie są klasycznymi strukturami deterministycznymi, takimi jak *if...then*, *case* itd. Zasadniczym źródłem niedeterminizmu jest instrukcja selekcji (*select*) umożliwiająca niedeterministyczne oczekiwanie na komunikację z innymi zadaniami. Syntaktyka instrukcji przypomina nieco postać instrukcji wyboru

```
select
  when B1  $\Rightarrow$  A1;
  or when B2  $\Rightarrow$  A2;
  .....
  or when Bn  $\Rightarrow$  An;
  else S
end select;
```

gdzie *B1*, ..., *Bn* są wyrażeniami boolowskimi, zaś *A1*, ..., *An*, *S* ciągami instrukcji. Wyrażenie boolowskie, które występuje w warunku *when* może zawierać tylko te zmienne, które są dostępne dla zadania wykonującego

daną instrukcję selekcji. Alternatywne ciągi instrukcji $A_1, \dots, A_n$ są ciągami, w których pierwszą instrukcją musi być instrukcja akceptacji (accept - p. 5.3) albo instrukcja opóźniająca (delay - p. dalej). Ciąg instrukcji S jest dowolny. W szczególności, gdy warunki $B_1, \dots, B_n$ są takie, że zawsze co najmniej jeden z nich jest prawdziwy, ciąg ten może być opuszczony. Podobnie, gdy warunek B_i jest tożsamościowo prawdziwy, wtedy możemy go także pominąć, pisząc

..... or A_i ;

Wykonanie instrukcji selekcji jest opisane przez podane niżej reguły. Dla zwięzłości będziemy mówić, że alternatywa A_i jest otwarta, jeżeli skojarzony z nią warunek B_i jest prawdziwy.

1. Oblicza się wszystkie warunki $B_1, \dots, B_n$ i określa zbiór alternatyw otwartych.

2. Otwarta alternatywa rozpoczynająca się instrukcją akceptacji może być wykonana, jeżeli zachodzi skojarzenie z odpowiadającym jej wywołaniem procedury wejściowej w innym procesie.

3. Otwarta alternatywa rozpoczynająca się instrukcją opóźnienia może być wykonana, jeżeli przed upływem odcinka czasu wskazanego przez tę instrukcję nie została wykonana żadna inna alternatywa.

4. Jeżeli żadna z otwartych alternatyw nie może być wykonana natychmiast i istnieje warunek else, to zostaje wykonany ciąg instrukcji S . Jeżeli warunek else nie istnieje, to zadanie oczekuje aż jedna z otwartych alternatyw będzie mogła być wykonana dzięki spełnieniu warunków opisanych regułą 2 lub 3.

5. Jeżeli zbiór alternatyw otwartych jest pusty i istnieje warunek else, to wykonywany jest ciąg instrukcji S . Jeżeli warunek else nie występuje, to pojawia się zastrzeżenie "błąd selekcji".

Ciało poprzednio rozważanego zadania bufor można obecnie przedstawić w pełnej postaci:

```
task body bufor is
  rozmiarbuf: const integer := 10;
  buf: array (1..rozmiarbuf) of pakiet;
  dox,zx: integer range 1..rozmiarbuf := 1;
  licznik: integer range 0..rozmiarbuf := 0;
begin
  loop
    select
      when licznik < rozmiarbuf =>
        accept pisz(e: in pakiet) do
          buf(dox) := e
        end pisz;
```



```

    dox := dox mod rozmiarbuf + 1;
    licznik := licznik+1;
or when licznik > 0 ⇒
    accept czytaj (v: out pakiet) do
        v:=buf(zx);
    end czytaj;
    zx:=zx mod rozmiarbuf-1;
    licznik i= licznik-1;
end select
end loop
end bufor;

```

Zadanie bufor zarządza cykliczną tablicą ze zmiennymi *dox*, *zx* wskazującymi końcową i początkową pozycję tablicy. Warunek *licznik* < *rozmiarbuf* stanowi pierwszy warunek instrukcji selekcji i zapobiega przed przepełnieniem tablicy bufor przez operację *pisz*. Drugi warunek *licznik* > 0 chroni tablicę bufor przed dostępem w sytuacji, gdy nie zawiera on żądanych danych. Jeżeli < 0 < *licznik* < *rozmiarbuf* i jednocześnie pojawia się wywołania portów *pisz* oraz *czytaj*, to wybór jednej z alternatyw instrukcji selekcji jest niezdeterminowany.

Wspomniana poprzednio instrukcja opóźniająca ma postać

delay (<wyrażenie liczbowe>)

i powoduje opóźnienie wykonywania zadania o odcinek czasu określany wartością swego argumentu. Wyrażenie liczbowe określa całkowitą liczbę jednostek czasu przyjętych w danej implementacji języka. Instrukcję delay można wbudować do instrukcji selekcji. Rozważając ostatnio przedstawiony przykład, zastosowaną w nim instrukcję selekcji możemy rozbudować do postaci

```

select
    when licznik < rozmiarbuf ⇒
        .....
    or when licznik > 0 ⇒
        .....
    or when delay 10.0*minuta;
        initiate testsystemu;
end select;

```

W tej postaci instrukcja powodowałaby, że w razie braku wywołania jednej z procedur *pisz* lub *czytaj* przez okres 10 minut nastąpiłaby inicjalizacja zadania diagnostycznego o nazwie *testsystemu*.

6.5. Przykładowe programy

Klasyczny przykład Dijkstry o filozofach siedzących za okrągłym stołem i jedzących spaghetti ma następujące rozwiązanie:

```

task filozof (1..5) is
end filozof;
task stół is
  entry zajmij (i: in 1..5);
  entry zwolnij (j: in 1..5);
end stół;
task body filozof is
begin
  loop
    .....
    stół.zajmij(filozof^indeks);
    ..... jedzenie .....
    stół.zwolnij(filozof^indeks);
    .....
  end loop
end filozof;
task body stół is
  jedzący: set of 1..5 :=[ ];
begin
  loop
    for i in 1..5 loop
      select
        when [i+1,i-1] *jedzący  $\Rightarrow$ 
          accept zajmij(i: in 1..5) do
            jedzący:=jedzący+ [i];
          end zajmij;
        or
          accept zwolnij(i: in 1..5) do
            jedzący:=jedzący-[i];
          end zwolnij;
        else
          null
        end select
      end loop
    end loop
  end stół;

```


Uwagi:

1. Program jest niepełny, gdyż zawiera tylko deklarację zadań, bez ich inicjalizacji.

2. Pominięto wyjaśnienie mechanizmu parametryzacji poszczególnych egzemplarzy rodziny zadań. Tu chodzi o sposób określenia wartości parametru filozof indeks w treści zadania filozof. Należy przyjąć, że parametr ten jest ustalony w momencie inicjalizacji konkretnego egzemplarza zadania i przyjmuje wartość równą indeksowi zadania w rodzinie filozof.

3. Instrukcja

```
for i in 1..5 loop
.....
end loop;
```

odpowiada pascalowskiej instrukcji

```
for i:=1 to 5 do .....;
```

4. Słowo null oznacza instrukcję pustą.

5. W rozwiązaniu zadania stół przyjęto aktywne oczekiwanie na zgłoszenia od zadań filozof. Oznacza to, że zadanie stół wykonuje bez przerwy obliczenia (najbardziej zewnętrzna pętla loop), w których testuje, czy istnieją wywołania jego portów (wykonywanie instrukcja selekcji select dla kolejnych wartości i=1,...,5 w pętli for).

Kolejny przykład stanowi implementację w języku Ada tablicy sortującej Brinch-Hansena.

```
task sortuj (1..n) is
  entry weź (c: in integer);
  entry daj (c: out integer);
end sortuj;
task body sortuj is
  x, z, n: integer;
begin
  loop
    accept weź (c: in integer) do
      x:=c
    end weź;
    n:=1;
  loop
    select
      accept weź (c: in integer) do
        z:=c
      end weź;
      if x <= z then
        sortuj (sortujindeks+1).weź(z);
```

```

    else
        sortuj(sortuj'indeks+1).weź(x);
        x:=z
    end if;
    n:=n+1;
or
    accept daj(c: out integer) do
        c:=x;
    end daj;
    n:=n-1;
    if n > 0 then
        sortuj(sortuj'indeks+1).daj(x);
    else
        exit
    end if
    end select;
end loop;
end loop;
end sortuj;

```

Uwagi:

- 1, 2 - jak w poprzednim przykładzie.
3. Instrukcja exit jest wyjściem z zewnętrznej pętli loop; wykonanie instrukcji kończy obliczenia pętli.

Ostatni przykład to znane sito Eratosthenesa wyznaczające liczby pierwsze:

```

task sito (1..100) is
    entry test (n: in integer);
end sito;
task body sito is
    p,mp,m: integer;
begin
    accept test (n: in integer) do
        p:=n;
    end test;
    drukuj(p);
    mp:=p;
    loop
        accept test (n: in integer) do
            m:=n;
        end test;

```



```

while mp < m loop
  mp := mp + p
end loop;
if mp > m then
  sito (sito indeks + 1).test(m)
end if;
end loop;
end sito;

```

Uwagi:

1,2 - jak w poprzednich przykładach.

ĆWICZENIA

1. Ułożyć program w języku Ada mnożący macierze zgodnie z algorytmem z p. 5.2.
2. Ułożyć program rozwiązujący problem przetwarzania telegramów (ćw. 2, rozdz. 4).
3. Zaproponować system buforowania komunikatów w węźle podsieci teletransmisyjnej (ćw. 3, rozdz. 3). Ułożyć program symulujący działanie węzła zgodnie z przyjętym systemem buforowania.
4. Ułożyć program symulujący współdziałanie węzłów podsieci teletransmisyjnej. Wykorzystać model węzła z ćw. 3.
5. Ułożyć program rozwiązujący problem czytelników i pisarzy (ćw. 1, rozdz. 2).
6. Zaprogramować w języku Ada rozwiązanie problemu z ćw. 1, rozdz. 4.
7. Napisać w języku Ada szkielet programu, w którym procesy ubiegające się o dostęp do jednorodnych zasobów są zarządzane zgodnie z algorytmem bankiera (zad. 7, rozdz. 3).
8. Przeanalizować możliwość zastąpienia mechanizmów komunikacji w językach procesów komunikujących się Hoare'a i procesów rozproszonych Brinch Hansena odpowiednimi konstrukcjami w języku Ada.
9. Przeanalizować możliwość zastąpienia mechanizmu dynamicznego tworzenia i modyfikacji sieci procesora Kahna odpowiednimi konstrukcjami w języku Ada.

7. MECHANIZMY RÓWNOLEGŁOŚCI W JĘZYKU PROGRAMOWANIA CHILL

7.1. Wprowadzenie

Język programowania Chill, podobnie jak język Ada, powstał w wyniku szeroko zakrojonej międzynarodowej współpracy. Patronat nad tymi pracami sprawował Międzynarodowy Doradczy Komitet Telefonii i Telegrafii (franc. Comité Consultatif International Télégraphique et Telephonique - CCITT) - organ Międzynarodowego Związku Telekomunikacyjnego (franc. Union International des Telecommunications - UIT). Nazwa CHILL jest akronimem od CCITT High Level Language. Początkowym zamierzeniem projektu nowego języka programowania było stworzenie narzędzia przeznaczonego do programowania elektronicznych centrali telefonicznych. W trakcie prac okazało się jednak, że definiowany język jest na tyle ogólny, że może być z powodzeniem wykorzystywany do innych aplikacji telekomunikacyjnych (np. programowanie węzłów podsieci transmisji danych w sieciach komputerowych, symulacja dyskretna), a także do obliczeń uniwersalnych.

Prace nad językiem były prowadzone w latach 1974-1980, z udziałem prawie 30 organizacji z ponad 20 krajów, i ostatecznym ich wynikiem było opracowanie dokumentu standaryzacyjnego [71] stanowiącego opis języka. CHILL osiągnął więc status międzynarodowego standardu i będzie, zwłaszcza w telekomunikacji, niewątpliwym konkurentem języka Ada, który dopiero ubiega się o uzyskanie statusu standardu w ramach Międzynarodowej Organizacji Normalizującej (ang. International Standard Organization - ISO). Oba języki mają porównywalny "stopień złożoności". Ze względu na to, że język Chill bazuje na modelu mieszanego środowiska ma on w stosunku do Ady bardziej rozbudowane mechanizmy obliczeń równoległych.

Omawiane w bieżącym rozdziale elementy języka Chill są przedstawiane w sposób nieformalny. Świadomie zmodyfikowano pisownię, wprowadzając podkreślanie słów kluczowych, zrezygnowano z wyjaśniania niektórych pojęć - takich jak np. location, mode - zastępując je najbardziej bliskoznacznymi odpowiednikami z języka Pascal czy Concurrent Pascal, dokonano pewnych uproszczeń, nieistotnych z punktu widzenia mechanizmów równoległości.

W całym rozdziale przyjęto konwencję, że wszystkie elementy opcjonalne (tzn. takie, które mogą, ale nie muszą wystąpić) w definicjach jednostek syntaktycznych są ujmowane w nawiasy kwadratowe [,], a więc np. zapis $A[B]C$ oznacza dwa ciągi AC oraz ABC.

7.2. Struktura programów

Jednostkami strukturalizacji programów są bloki, procedury, procesy, moduły i regiony. Bloki i procedury są - w przybliżeniu - odpowiednikami bloków i procedur w Pascalu, odpowiednikiem procesu są zadania w Adzie. Moduły pełnią rolę pewnej przegrody, dzięki której można rozszerzać dostępności obiektów lokalnych modułu, lub też definiować znaczenie obiektów nielokalnych, wykorzystywanych w module. W tym zakresie moduły spełniają rolę podobną do modułów w języku Modula. Regiony natomiast - to odpowiedniki monitorów w Concurrent Pascalu. W dalszej części nie definiuje się w pełni wymienionych tu jednostek strukturalizacji; ich znaczenie ukazuje się przez komentarze do przedstawionych dalej przykładów.

Procesy - jako jednostka strukturalizacji najbardziej istotna z rozważanego punktu widzenia - są definiowane dwuetapowo. Etap pierwszy odpowiada definicji typu procesu, etap drugi - kreacji konkretnego egzemplarza procesu. Definicja typu procesu ma postać:

NP : process (PF);

DL;

SL

end

gdzie NP - nazwa procesu, PF - lista parametrów formalnych, DL - ciąg deklaracji lokalnych obiektów procesu oraz SL - ciąg instrukcji. Parametry formalne PF są wykorzystywane przy kreacji i inicjalizacji procesu; tu dla uproszczenia przyjmujemy, że komunikacja tych parametrów, przy kreacji procesu, odbywa się przez podstawienie wartości parametrów aktualnych pod odpowiednie parametry formalne. Obiektami, które mogą być deklarowane w ciągu DL są m.in.: typy, procedury, procesy (procesy mogą być zagnieżdżane) oraz sygnały.

Kreacja i inicjalizacja egzemplarza procesu ma postać:

start NP (PA).

Dopuszcza się kreację dowolnej liczby egzemplarzy procesu danego typu. Dynamicznie wykreowane procesy są odróżnialne od siebie przez to, że każdemu z nich jest przypisany niejawnie pewien identyfikator (adres obszaru pamięci przydzielony procesowi). Identyfikator taki można zapamiętywać w zmiennych o specjalnie wydzielonym typie adresowym (instance mode). Jeżeli np. A jest zmienną takiego typu, to instrukcja postaci:

A := start NP(PA)

poza utworzeniem i inicjalizacją procesu typu NP spowoduje podstawienie

pod zmienną A adresu tego procesu. Proces może także poznać swój identyfikator podczas obliczeń, używając stałej this, dostarczającej adresu tego procesu, w którym jest użyta.

Uruchomiony proces rozpoczyna ciąg obliczeń, który przebiega równolegle z obliczeniami innych procesów. Proces kończy się, gdy osiągnie koniec ciągu SL. Jeżeli podczas obliczeń procesu napotka on na instrukcję zatrzymania stop, następuje wstrzymanie dalszych obliczeń i proces zakończy się, gdy skończą się wszystkie inne procesy wykreowane przez ten proces.

Równolegle pracujące procesy mogą komunikować się ze sobą za pośrednictwem wspólnego środowiska bądź pewnych mechanizmów komunikacji w środowisku rozproszonym. W pierwszym przypadku komunikacja odbywa się z wykorzystaniem tzw. regionów, w drugim zaś komunikacja może odbywać się dwoma sposobami: z wykorzystaniem buforów i sygnałów.

7.3. Komunikacja we wspólnym środowisku

Komunikacja we wspólnym środowisku wymaga istnienia mechanizmów zapewniających wykluczający dostęp procesów do wspólnych danych. Jednostkami strukturalnymi mającymi taki mechanizm są tzw. regiony. Regiony są kolekcją wspólnych zmiennych, do których można uzyskiwać dostęp wyłącznie poprzez wywoływanie procedur krytycznych (odpowiednik procedur wejściowych monitora) regionu. Jeżeli region jest wolny - tzn. nie jest zajęty wykonywaniem swych procedur krytycznych - to przyjmuje wywołania od procesów; spośród jednocześnie napływających wywołań region wybiera dowolne i wykonuje je, podczas gdy pozostałe procesy są zawieszane. Po wykonaniu wywołanej procedury krytycznej region wybiera dowolny spośród oczekujących procesów i wykonuje wskazaną przez niego procedurę krytyczną. Opisany mechanizm krótkoterminowego szeregowania procesów jest zatem taki sam, jak w monitorach Concurrent Pascala.

Dodatkowo regiony mają mechanizm umożliwiający szeregowanie średnioterminowe. Mechanizm opiera się na pojęciu zmiennej zdarzeniowej oraz na operacjach, jakie można na niej wykonywać. Deklaracja zmiennej E typu zdarzeniowego ma postać

dcl E event [(d)],

gdzie d jest dodatnim wyrażeniem całkowitoliczbowym oznaczającym długość zmiennej zdarzeniowej E; brak opcji d oznacza przypisanie zmiennej długości równej jeden. Zmienna E służy do zapamiętywania identyfikatorów zawieszanych procesów; długość zmiennej oznacza maksymalną liczbę procesów, które mogą być przez nie zapamiętane. Jedynymi operacjami, jakie można wykonywać na zmiennych zdarzeniowych, są operacje continue,

delay oraz delay case. Pierwsza z nich służy odwieszaniu procesów, a dwie pozostałe zawieszaniu procesów. Dokładne ich znaczenie jest następujące:

Operacja kontynuacji, continue E, powoduje reaktywowanie procesu o najwyższym priorytecie spośród zawieszonych procesów, pamiętanych w E. Identyfikator reaktywowanego procesu jest usuwany z E. Jeżeli więcej niż jeden proces ma najwyższy priorytet, to wybór jednego z nich jest nie-deterministyczny. Jeżeli zmienna E jest pusta - nie pamięta żadnych zawieszonych procesów - to operacja jest równoważna instrukcji pustej.

Operacja zawieszenia

delay E [priority p],

gdzie p jest nieujemnym wyrażeniem całkowitoliczbowym, powoduje zawieszenie procesu, który spowodował jej wykonanie i zapamiętanie jego identyfikatora oraz priorytetu p w zmiennej zdarzeniowej E. Brak priorytetu p w operacji powoduje przyporządkowanie procesowi domyślnej wartości priorytetu równej zeru. Zawieszony proces może być reaktywowany dopiero po wykonaniu operacji continue E. Jeżeli w momencie wykonania operacji delay liczba zapamiętanych zawieszonych procesów jest równa długości zmiennej zdarzeniowej, to nastąpi błąd wykonania programu.

Operacja zawieszenia alternatywnego ma postać:

delay case [set A] [priority p]

(LE₁): SL₁

- - - - -

(LE_n): SL_n

esac

gdzie A jest zmienną typu adresowego, p ma znaczenie takie, jak przy poprzedniej operacji, LE₁, ..., LE_n są listami zmiennych zdarzeniowych, zaś SL₁, ..., SL_n są ciągami instrukcji. Wykonanie tej operacji powoduje zawieszenie procesu, przy czym jego identyfikator oraz priorytet zostają zapamiętane we wszystkich zmiennych zdarzeniowych występujących w listach LE₁, ..., LE_n. Zawieszony proces będzie odwieszony, jeżeli zostanie wykonana taka operacja continue, na jednej spośród wymienionych zmiennych zdarzeniowych, która wybierze ten proces do reaktywowania. Reaktywowany proces wykona ten spośród ciągów instrukcji SL₁, ..., SL_n, który odpowiada liście LE zawierającej tę zmienną zdarzeniową, na której wykonano operację kontynuacji. W razie gdy zmienna, na której wykonano operację znajduje się na kilku listach, wówczas wybór odpowiedniego ciągu instrukcji jest niedeterministyczny. Po odwieszeniu procesu jego identyfikator zostaje usunięty ze wszystkich wymienionych zmiennych zdarzeniowych. Jeżeli w momencie zawieszenia procesu zostaje przekroczona dłu-

gość którejkolwiek ze zmiennych zdarzeniowych, następuje błąd wykonania programu. Opcja set A służy do tego, aby odwieszony proces mógł zapamiętać w zmiennej A identyfikator tego procesu, który spowodował jego odwieszenie.

Poniżej przedstawiony przykład ilustruje zastosowanie konstrukcji regionu oraz zmiennych zdarzeniowych do zorganizowania wykluczającego dostępu do wspólnej puli jednorodnych zasobów.

```

1   przydział_zasobów : region
2       grant przydziel,zwolnij;
3       newmode zbiór_zasobów=int(0:9);
4       dcl przydzielony array (zbiór zasobów) bool:=
           (: (zbiór zasobów): false);
5       dcl zwolnienie zasobu event;
6       przydziel : proc( )(int);
7           do for ever;
8               do for i in zbiór_zasobów;
9                   if not przydzielony (i) then przydzielony (i):=true;
10                  return (i);
11                  fi;
12              od;
13              delay zwolnienie_zasobu
14          od;
15       end przydziel;
16       zwolnij : proc (i int);
17           przydzielony (i) :=false;
18           continue zwolnienie_zasobu;
19       end zwolnij;
20   end przydział_zasobów

```

Przykład wprowadza jednocześnie nie omawiane dotychczas elementy języka, które wymagają dodatkowych wyjaśnień. Wiersz 2 służy do ustalenia reguł dostępu: słowo grant (odpowiednik słowa define w języku Modula) oznacza, że obiekty lokalne przydziel oraz zwolnij udostępnia się na zewnątrz regionu (w bloku bezpośrednio otaczającym region). W wierszu 3 wprowadza się definicję nowego typu (newmode), którego zbiorem wartości są liczby całkowite (int) z przedziału [0,9]. Wiersze 4, 5 są deklaracją (dcl) zmiennych: Wiersz 4 deklaruje zmienną typu tablicowego (array), której elementy są typu boolowskiego (bool), oraz inicjalizuje tę zmienną, przypisując wartości false wszystkim jej elementom.

Wiersze 6-15 oraz 16-19 są definicjami procedur (proc) krytycznych regionu. Parametry procedur są rozdzielone na dwie grupy: parametry wejściowe - pierwsza para nawiasów, parametry wyjściowe - druga para. Pro-

cedura przydziel ma pusty zbiór parametrów wejściowych i jeden parametr wyjściowy, a procedura zwolnij - jeden parametr wejściowy i brak parametrów wyjściowych.

Z instrukcji występujących w treści procedur komentarza wymagają tylko trzy: instrukcja do for ever ... od jest nieskończoną pętlą; instrukcja return (i) oznacza zakończenie procedury (wyjście z pętli) oraz podstawienie wartości i pod parametr wyjściowy; instrukcja do for i in zbiór_zasobów jest instrukcją pętli, przy czym ciekawostką jest to, że i stanowi tutaj tylko parametr oraz to, że kolejność przyjmowania przez i wartości z przedziału [0,9] jest niezdeterminowana.

7.4. Komunikacja z wykorzystaniem sygnałów

Sygnały są obiektami, które mają służyć synchronizacji równoległe działających procesów podczas wzajemnego przekazywania informacji. Sygnały muszą być deklarowane w pewnym bloku na tym samym poziomie, co wykorzystujące je procesy (sygnały są zatem w procesach obiektami nielokalnymi). Postać deklaracji sygnału S jest następująca:

signal S [= (T₁, ..., T_n)] [to P],

gdzie T₁, ..., T_n są identyfikatorami typów, P - identyfikatorem procesu.

Opcja (T₁, ..., T_n) służy do zdefiniowania zbioru wartości, które mogą być przesyłane wraz z sygnałem. Opcja to P oznacza, że jedynym odbiorcą sygnału jest proces o nazwie P.

Na sygnałach można wykonywać dwie operacje: operację przesyłania sygnału (send) oraz operację odbioru sygnału (receive case).

Operacja przesyłania sygnału S ma postać

send S [(W₁, ..., W_n)] [to WP] [priority p],

gdzie W₁, ..., W_n są wartościami, odpowiednio typu T₁, ..., T_n; WP jest wyrażeniem, którego wartością jest identyfikator pewnego procesu; p - nieujemnym wyrażeniem całkowitym oznaczającym priorytet.

Brak opcji W₁, ..., W_n oznacza, że z sygnałem nie przesyła się wartości, brak opcji to WP oznacza, że sygnał jest kierowany do wszystkich procesów, brak zaś opcji priority p oznacza, że sygnałowi przypisuje się domyślnie wartość priorytetu równą zeru. Operacja send wykonana w pewnym procesie przesyła sygnał wraz z dołączoną listą wartości i priorytetem do pewnego wskazanego lub do wszystkich procesów, i powoduje ewentualne reaktywowanie procesu, który został zawieszony i oczekuje na ten sygnał.

Operacja odbioru sygnałów ma postać:

```
receive case [set A]
  (S1 [in LN1]):SL1
  .....
  (Sn [in LNn]):SLn
  [else SL0]
esac,
```

gdzie A jest zmienną typu adresowego; S₁,...,S_n są nazwami różnych sygnałów; LN₁,...,LN_n są listami identyfikatorów parametrów; SL₀,...,SL_n są ciągami instrukcji.

Lista identyfikatorów parametrów LN₁ występuje wówczas, gdy sygnał S₁ jest przesyłany wraz z listą pewnych wartości; parametry z tej listy służą wtedy do zapamiętywania przesłanej listy wartości. Znaczenie operacji jest następujące: jeżeli w momencie rozpoczęcia realizacji operacji oczekuje na odbiór jeden spośród sygnałów S₁,...,S_n, to zostaje wybrany ten, który ma najwyższy priorytet, (gdy jest kilka takich sygnałów, wtedy wybór jednego z nich jest niedeterministyczny), pod odpowiadającą mu listę parametrów zostają podstawione przesyłane z sygnałem wartości, a następnie wykonuje się odpowiadający wybranemu sygnałowi ciąg instrukcji. Jeżeli w momencie rozpoczęcia realizacji nie ma oczekującego sygnału oraz nie ma opcji else, to proces wykonujący operację zawieszają się, aż do momentu nadejścia jednego z oczekiwanych sygnałów; gdy natomiast opcja else występuje, wtedy wykonuje się odpowiadający jej ciąg instrukcji SL₀. Sygnał może być odebrany przez proces tylko wówczas, gdy:

- jeżeli podczas operacji send występuje opcja to WP, to wartość WP wskazuje ten proces,
- jeżeli w deklaracji sygnału występuje opcja to P, to P jest nazwą tego procesu.

Zmienna adresowa A służy do zapamiętania nazwy tego procesu, od którego odebrano sygnał; jeśli zmienna ta występuje, to podstawienie pod nią odpowiedniej wartości dokonuje się przed obliczeniem ciągów SL_i.

Zastosowania mechanizmu sygnałów pokazuje następujący fragment programu: Przedstawiony moduł zarządca składa się m.in. z pewnej puli procesów (oblicz), z których usług będą korzystać inne procesy, zewnętrzne w stosunku do modułu (zarządca). Wewnątrz modułu są używane sygnały (inicjuj) i (zakończ), wymieniane pomiędzy procesem (alokator) a procesami (oblicz). Pomiedzy modułem a jego otoczeniem są wymieniane sygnały (przydziel), (zwolnij), (wykonaj), (przeciążony), (gotowy), (odczytaj);


```

1  zarządca : module
2    seize przydziel, zwolnij, wykonaj, przeciążony, gotowy, odczytaj;
3    signal inicjuj=(instance), zakończ;
4    alokator:process( );
5    newmode liczba=int (1:100);
6    dcl liczniki array (liczba)
7        struct(nr instance,
8                status set(wolny,zajęty));
9    do for in liczniki;
10       k:=(start oblicz( ),wolny:);
11    od;
12    do for ever;
13    begin
14       dcl użytkownik instance
15       oczekiwane_sygnały:
16       receive case set użytkownik;
17       (przydziel):do for k in liczniki;
18           do with k;
19               if status=wolny
20               then status:=zajęty;
21               send inicjuj(użytkownik) to oblicz;
22               exit oczekiwane_sygnały;
23           fi;
24       od;
25       od;
26       send przeciążony to użytkownik;
27       (zwolnij in nr-oblicz):
28       send zakończ to nr-oblicz;
29       znajdź_oblicz:
30       do for k in liczniki;
31           do with k;
32               if nr-oblicz=nr then
33                   status:=wolny;
34               exit znajdź_oblicz;
35           fi;
36       od;
37       od znajdź_oblicz;
38   esac oczekiwane_sygnały;
39 end;
40 od;
41 end alokator;

```

```

42  oblicz : process( );
43  do for ever;
44  begin
45      dcl użytkownik instance,
46          licz int:=0;
47      receive case
48          (inicjuj in odbiorca):
49          send gotowy to odbiorca;
50          użytkownik:=odbiorca;
51      esac;
52      robocza_pętla:
53      do for ever
54          receive case
55              (wykonaj):licz:=licz+1;
56              (zakończ):send odczytaj (licz) to użytkownik;
57              exit robocza_pętla;
58          esac;
59          od robocza_pętla;
60      end;
61      od;
62      end oblicz;
63      start alokator;
64  end zarządca;

```

W przykładzie występują nowe, nie wyjaśnione dotychczas konstrukcje językowe. Wiersz 2 wprowadza listę obiektów, wykorzystywanych w module zarządca, a które są zdefiniowane na zewnątrz tego modułu. Łatwo wywnioskować z treści modułu, że definicja tych obiektów jest postaci:

```

signal przydziel,zwolnij=(instance),
    wykonaj,przeciążony,gotowy,
    odczytaj =(int);

```

Występujące tutaj, a także w wierszach 3, 7, 14, 45, słowo instance oznacza typ adresowy, służący do identyfikacji procesów. Wiersze 6, 7, 8 są deklaracją zmiennej tablicowej liczniki, której elementy są strukturami - w terminologii języka Pascal rekordami - o dwóch polach nr oraz status. Wiersze 9, 10, 11 są odmianą instrukcji iteracji; kolejne iteracje dotyczą elementów tablicy liczniki. Instrukcje exit występujące w wierszach 22, 34, 57 są instrukcjami skoku odpowiednio do wierszy 39, 38, 60 będących pierwszymi wierszami bezpośrednio za końcem bloków, które są wymienione jako argumenty instrukcji. Instrukcje do with występujące w wierszach 18, 31 są odpowiednikami instrukcji wiążącej w Pascalu.

7.5. Komunikacja z wykorzystaniem buforów

Bufory stanowią kolejny obiekt służący komunikacji i synchronizacji procesów. Bufory, podobnie jak sygnały, muszą być deklarowane na tym samym poziomie, na którym deklarowane korzystające z nich procesy. Bufory są deklarowane jako zmienne typu buforowego. Definicja typu buforowego ma postać:

buffer[(d)] T,

gdzie d jest wyrażeniem całkowitym, nieujemnym, zwanym długością bufora, zaś T jest typem danych, które mogą być przechowywane w tym buforze. Brak opcji d oznacza, że długość bufora wynosi zero. Na zmiennych typu buforowego, np. zmiennej zadeklarowanej w postaci

dcl B buffer (d) T

można wykonywać operacje: przesłania do bufora (send), odbioru z bufora (receive) i odbioru alternatywnego (receive case).

Operacja przesłania wartości wyrażenia w, typu T do bufora B ma postać

send B (w)[priority p].

Operacja działa w sposób następujący: Jeżeli aktualna liczba wiadomości przechowywana w buforze B jest mniejsza od długości bufora, to wartość wyrażenia w oraz priorytet p zostają zapamiętane w buforze, zaś proces, który wykonał tę operację kontynuuje swe dalsze obliczenia. Jeżeli natomiast aktualna liczba wiadomości przechowywana w B jest równa jego długości, wtedy proces wykonujący operację zostaje zawieszony. Zawieszenie trwa dopóty, dopóki nie pojawi się wolne miejsce w buforze, bądź nastąpi odbiór wartości w przez jeden z oczekujących na nią procesów. Brak opcji priority powoduje przyporządkowanie wiadomości priorytetu zerowego.

Operacja odbioru z bufora B ma postać

receive B

Operację można traktować jako procedurę funkcyjną, która działa następująco: Spośród wiadomości zawartych w buforze B oraz tych wiadomości, które pewne zawieszone procesy starają się przekazać do bufora B (procesy wykonujące operację send), wybiera się wartość odpowiadającą wiadomości o najwyższym priorytecie. Jeżeli bufor B nie zawiera żadnych wiadomości i nie ma odpowiednich zawieszonych procesów, to proces wykonujący operację receive zawiesza się, aż do momentu przesłania do B pewnej wiadomości.

Operacja odbioru alternatywnego ma postać:

```

receive case [set A;]
  (B1 in N1) : SL1
  .....
  (Bn in Nn) : SLn
  [else SL0]
esac,

```

gdzie B₁,...,B_n są nazwami buforów; N₁,...,N_n - nazwami parametrów; SL₀, SL₁,...,SL_n są ciągami instrukcji, zaś A jest zmienną adresową.

Operacja polega na odbiorze pojedynczej wiadomości z jednego z buforów B₁,...,B_n, lub od jednego z zawieszonych procesów oczekujących na przesłanie wiadomości do tych buforów, zapamiętaniu tej wiadomości poprzez podstawienie pod jeden z parametrów N₁,...,N_n, a następnie na wykonaniu jednego z ciągów instrukcji SL₁,...,SL_n odpowiadającego wybranemu buforowi. Dokładniej, wykonanie operacji przebiega następująco: w niezdeteminowanej kolejności są przeglądane bufor B₁,...,B_n. Jeżeli przeglądany bufor, powiedzmy B_i, ma zapamiętaną pewną wiadomość lub istnieją pewne zawieszone procesy przesyłające wiadomość do tego bufora, to spośród tych wiadomości zostaje wybrana wiadomość o najwyższym priorytecie, a związana z nią wartość zostaje podstawiona pod parametr N_i, a następnie wykonany ciąg instrukcji SL_i. Jeżeli w wyniku przeglądu okaże się, że z żadnym buforem nie są skojarzone wiadomości do odbioru, to dalsze wykonanie operacji zależy od opcji else. Jeżeli opcja ta istnieje, to jest wykonywany ciąg instrukcji SL₀, a jeśli jest jej brak, to proces wykonujący operację zostaje wstrzymany, aż do momentu, gdy do jednego z buforów B₁,...,B_n zostanie skierowana pewna wiadomość. Opcja set A, tak jak w przypadkach poprzednich, służy do zapamiętania identyfikatora tego procesu, z którego odebrano wiadomość.

Kolejny przykład ilustruje zastosowanie omawianego mechanizmu procesów i jednocześnie wprowadza dalsze szczegóły języka Chill. Przykład jest modyfikacją programu z poprzedniego punktu, polegającą na zastąpieniu komunikacji za pomocą sygnałów komunikacją za pomocą buforów.

```

1  otoczenie_użytkowników module
2  grant bufory_użytkownika,wiadomości_alokatora,
3      bufory_alokatora,wiadomości_oblicz,
4      bufory_oblicz;
5  newmode
6      wiadomości_użytkownika=
7      struct (type set (przeciążony,gotowy,odczytaj,
8                          identyfikator_alokatora),
9      case typ of

```



```

10         (przeciążony):
11         (gotowy): oblicz ref bufory_oblicz,
12         (odczytaj): licz int,
13         (identyfikator_alokatora):
14         alokator ref bufory_alokatora
15         esac),
16     bufory_użytkownika=buffer(1) wiadomości_użytkownika,
17     wiadomości_alokatora=
18     struct (typ set (przydziel,zwolnij,ident oblicz) ,
19     case typ of
20         (przydziel): użytkownik ref bufory_użytkownika,
21         (zwolnij,ident_oblicz):
22             oblicz ref bufory_oblicz
23     esac),
24     bufory_alokatora=buffer(1) wiadomości_alokatora,
25     wiadomości_oblicz=
26     struct (typ set (inicjuj,wykonaj,zakończ),
27     case typ of
28         (inicjuj): użytkownik ref bufory_użytkownika,
29         (wykonaj,zakończ):
30     esac),
31     bufory_oblicz=buffer(1) wiadomości_oblicz,
32     dcl bufor_użytkownika bufory_użytkowników,
33     bufor_alokatora ref bufory_alokatora,
34     bufor_oblicz ref bufory_oblicz;
35     start alokator (→bufor_użytkownika);
36     bufor_alokatora:=(receive bufor_użytkownika). alokator;
37 end otoczenie_użytkownika;
38 zarządca:module;
39 seize bufory_użytkownika, wiadomości_alokatora,
40     bufory_alokatora, wiadomości_oblicz,
41     bufory_oblicz;
42 alokator:process (buf ref bufory_użytkownika);
43     dcl bufor_alokatora bufory_alokatora;
44     newmode nr_oblicz=int(1:10);
45     dcl liczniki array (nr_oblicz)
46         struct (nr ref bufory_oblicz,
47             status set (wolny,zajęty)),
48     wiadomość wiadomości_alokatora;
49     send buf→([identyfikator_alokatora,
50         →bufor_alokatora]);

```

```

51  do for K in liczniki;
52      start oblicz ( $\rightarrow$  bufor_alokatora);
53      K := [receive bufor_alokatora).ident_oblicz, wolny];
54  od;
55  do for ever
56      begin
57          dcl użytkownika ref bufory_użytkownika;
58          wiadomość := receive bufor_alokatora;
59          zarządzanie_wiadomościami:
60          case wiadomość.typ of
61              (przydziel):
62                  użytkownik := wiadomość.użytkownik;
63                  do for K in liczniki;
64                      do with K;
65                          if status = wolny then
66                              status := zajęty
67                              send nr  $\rightarrow$  ([inicjuj, użytkownik]);
68                              exit zarządzanie_wiadomościami
69                          fi;
70                      od;
71                  od;
72                  send użytkownik  $\rightarrow$  ([przeciążony]);
73              (zwolnij):
74                  send wiadomości.nr ([zakończ]);
75                  znajdź_oblicz:
76                  do for K in liczniki;
77                      do with K;
78                          if wiadomość.oblicz = nr
79                              then status := wolny
80                              exit znajdź_oblicz
81                          fi
82                      od
83                  od znajdź_oblicz;
84          esac zarządzanie_wiadomościami
85      end;
86  od;
87  end alokator;
88  oblicz : process (buf ref bufory_alokatora);
89      dcl bufor_oblicz bufory_oblicz;
90      send buf  $\rightarrow$  ([ident_oblicz,  $\rightarrow$  bufor_oblicz]);
91      do for ever

```



```

92   begin
93     dcl użytkownik ref bufory-użytkownika;
94       licz int := 0,
95       wiadomość wiadomości_oblicz;
96     wiadomość := receive bufor-oblicz;
97     case wiadomość.typ of
98       (inicjuj): użytkownik := wiadomość.użytkownik;
99         send użytkownik → ([gotowy, → bufor_oblicz]);
100      else (* sytuacja błędna *)
101      esac;
102     pętla_robocza:
103     do for ever
104       wiadomość := receive bufor_oblicz;
105       case wiadomość.typ of
106         (wykonaj): licz := licz + 1;
107         (zakończ):
108           send użytkownik → ([odczytaj,licz]);
109           exit pętla_robocza;
110         else (* sytuacja błędna *)
111         esac;
112       od pętla_robocza;
113     end;
114   od
115   end oblicz;
116   end zarządca;

```

Nowe konstrukcje językowe, które pojawiły się w przykładzie to: struktury (rekordy) z wariantami - wiersze 7-14, 18-23, 26-30; specjalny typ referencyjny (wskaźnikowy). Oznaczenie elementów struktur jest podobne do odpowiednich oznaczeń w Pascalu. Podstawianie wartości pod zmienną typu struktury z wariantami ma postać taką, jak przedstawiono w wierszach np. 49, 53, gdzie pierwszy element wewnątrz nawiasu kwadratowego oznacza wybór wariantu, a drugi wartość przyporządkowaną temu wariantowi. Typ referencyjny ref został w przykładzie użyty do identyfikacji buforów poprzez ich adresy. Znaczenie tego typu jest podobne do roli typu wskaźnikowego w Pascalu, np. wiersz 14

alokator ref bufory_alokatora

oznacza, że na polu alokator (struktury typu wiadomości użytkownika) będą występować adresy obiektów typu bufory alokatora. Jeżeli mamy zmienną typu niereferencyjnego np. bufor_alokatora (wiersz 43), to notacja

→bufor_alokatora

oznacza adres tej zmiennej (wiersz 52). Jeżeli natomiast mamy zmienną typu referencyjnego, np. parametr buf (wiersz 42), to notacja

buf→

oznacza zmienną określoną przez ten adres.

ĆWICZENIA

1. Napisać trzy wersje programu w języku Chill rozwiązujące zagadnienie czytelników i pisarzy (ćw. 1, rozdz. 2) i wykorzystujące odpowiednio mechanizmy komunikacji poprzez regiony, sygnały i bufor.

2. Wybierając dowolny mechanizm komunikacji napisać w języku Chill program rozdziału zasobów, zgodnie z algorytmem bankiera (ćw. 7, rozdz. 3), pomiędzy równoległe działające procesy.

3. Opracować model współpracy abonenta z centralą telefoniczną podczas nawiązywania połączenia. Napisać program w języku Chill, który będzie symulować tę współpracę. (Założyć, że centrala telefoniczna ma pewien proces zegara, który służy do wyznaczania bieżącego czasu.)

4. Wykorzystując model działania węzła podsieci teletransmisji danych (ćw. 3, rozdz. 3) oraz model współdziałania węzłów (ćw. 4, rozdz. 6) napisać odpowiednie programy symulujące w języku Chill.

5. Zaprogramować równoległy algorytm sortowania liczb i przedstawić jego realizację w języku Chill.

6. Dokładnie przeanalizować mechanizmy sygnałów i buforów w języku Chill. Czy mechanizmy te są wzajemnie zastępowalne?

7. Porównać mechanizmy komunikacji poprzez środowisko rozproszone w językach Ada i Chill. Czy mechanizmy te są wzajemnie zastępowalne?

8. OBLICZENIA SYNCHRONIZOWANE PRZEPŁYWEM DANYCH

8.1. Wprowadzenie

Rozwojowi technologii, która doprowadziła do miniaturyzacji i potaniaenia sprzętu komputerowego towarzyszy powstanie koncepcji nowych, odmiennych od tradycyjnych, von neumannowskich spojrzeń na programowanie. W zakresie programowania sekwencyjnego jest to coraz silniejszy rozwój programowania funkcyjnego (nieproceduralnego) - przykładem języka programowania opartego na tej koncepcji jest Prolog. W zakresie programowania równoległego jest to koncepcja obliczeń synchronizowanych przepływem danych - przykładem języka powstającego na bazie tej koncepcji jest VAL (Value Oriented Language) opracowywany w MIT pod kierunkiem Dennisa. W języku tym znajduje także odbicie koncepcja programowania funkcyjnego.

Programowanie sterowane przepływem danych formułuje program jako zbiór operacji (rozumianych tu intuicyjnie, na przykład jako działania arytmetyczne, logiczne itp.), który służy do realizacji zadania. Podkreślenia wymagają tu dwa aspekty:

- na operacje nie nakłada się z góry jawnego porządku ich wykonywania,
- dany zestaw wartości początkowych programu może nie wymagać wykonania wszystkich operacji programu.

Konwencja wyznaczania kolejności operacji oraz podzbioru wykorzystanych operacji wynika z potrzeby dostarczenia odpowiednich danych dla uzyskania wymaganych wyników. Opiera się ona na zasadzie, że daną operację można wykonać wówczas, gdy są gotowe (dostarczone lub obliczone) argumenty tej operacji. Obliczenia rozpoczynają się w chwili, gdy jest ustalony pewien zestaw danych początkowych; zestaw ten pozwala na uruchomienie realizacji tych operacji, które działają wyłącznie na danych początkowych. Uruchomione operacje są wykonywane niezależnie od siebie: po zakończeniu każdej z nich mogą powstać warunki do uruchomienia następnych operacji. Obliczenia rozpoczynają się i są kontynuowane do momentu, gdy nie istnieją operacje w trakcie wykonywania i nie ma innych operacji, dla których istnieją warunki ich uruchomienia.

W rozdziale przedstawiono uproszczoną wersję języka VAL [18] oraz omówiono krótko problemy architektury komputera dla efektywnej implementacji tego języka. To ostatnie zagadnienie - chociaż nie należy zasad-

niczo do zakresu skryptu - poruszono z uwagi na jego odmienność w stosunku do problemów architektury komputerów implementujących opisywane poprzednio podejścia.

8.2. Koncepcja języka programowania

Przedstawiony język SVAL (Simple VAL) jest uproszczoną wersją języka VAL, opracowanego w MIT w 1979 r. i stanowiącego kontynuację prac prowadzonych w tym ośrodku od wczesnych lat siedemdziesiątych [16,17,18]. Od języków tradycyjnych różni się on brakiem jawnego wyróżniania równoległości - równoległość obliczeń wynika z rozplywu danych w trakcie obliczeń. Język jest funkcyjnie zorientowany (value-oriented), co oznacza tutaj, że każda jednostka syntaktyczna definiuje pewną funkcję odwzorowującą zbiór wartości wejściowych w zbiór wartości wyjściowych. Przetworzeniu wielkości wejściowych w wyjściowe nie towarzyszą żadne efekty uboczne.

Język nie używa pojęcia komórki pamięci, a zatem i zmiennej (w sensie takim, jak w tradycyjnych językach programowania). Używa się natomiast pojęcia nazwy dla identyfikacji wartości definiowanych przez wyrażenia i funkcje - tak samo, jak jest to czynione w klasycznej matematyce. Funkcyjnie zorientowana semantyka języka umożliwia pisanie programów w postaci naturalnie wyrażającej "wrodzone" równoległości algorytmów.

Abstrakcyjnymi strukturami danych języka są typy: integer, real, boolean, character string (łańcuch znaków), structure (struktura) oraz procedure. Operacje, jakie można wykonywać na typach integer, real, boolean, character string są ogólnie znane i nie wymagają komentarza. Operacje wykonywane na strukturze będą zdefiniowane dalej (p. 8.3). Jedyną operacją, jaką można wykonać na procedurze, jest jej wywołanie.

Składnia języka jest na tyle prosta, że można przedstawić ją formalnie niemal w całości. Dla zwięzłości opisu przyjęto konwencję:

$$\begin{aligned} \{<E>\}^+ & \text{ oznacza } E \mid E, \{E\}^+ \\ \{<E>\} & \text{ oznacza } \{E\}^+ \mid \text{puste} \end{aligned}$$

```
<program> ::= program {<def procedury>} <wyrażenie> end
<def procedury> ::= <nazwa> = procedure (<lista param we>)
                        yields (<lista param wy>);
                        {<def procedury>};
                        <wyrażenie>
                        end <nazwa>
<lista param we> ::= {<deklaracja typu>}
<deklaracja typu> ::= <nazwa>:<typ>
```



```

<lista param wy> ::= {<typ>}
<wyrażenie> ::= <proste wyrażenie> | {<wyrażenie>}+ |
  <wyrażenie blokowe> | <wyrażenie warunkowe> |
  <wywołanie procedury>
<wyrażenie blokowe> ::=
  let{<deklaracja typu>};{<definicja nazwy>};
  in <wyrażenie> end
<definicje nazwy> ::= {<nazwa>} = <wyrażenie>
<wyrażenie warunkowe> ::=
  if <wyrażenie> then <wyrażenie>
  else <wyrażenie> end
<wywołanie procedury> ::= <nazwa>(<wyrażenie>)
<wyrażenie proste> ::= <stała> | <nazwa> |
  <prosty operator> <wyrażenie> |
  <wyrażenie> <prosty operator> <wyrażenie>
<typ prosty> ::= integer | real | boolean |
  character string | structure
<typ> ::= <typ prosty> | stream of <typ prosty>

```

Podstawową jednostką syntaktyczną jest definicja procedury, na przykład:

```

p = procedure (a1:T1,...,am:Tm)
  yields R1,...,Rn;
  <def procedury>,
  .....
  <def procedury>
  <wyrażenie>
end P

```

Procedura P ma m parametrów wejściowych a₁,...,a_m o typach, odpowiednio, T₁,...,T_m. Nazwy a₁,...,a_m muszą być różne od siebie i mogą występować w wyrażeniu stanowiącym treść procedury. Obliczenia procedury dostarczają uporządkowany n-elementowy zbiór wartości o typach R₁,...,R_n, stanowiący wynik wyliczenia wartości wyrażenia.

Pojęcie wyrażenia jest ogólniejsze od rozumienia tradycyjnego i może oznaczać wektor (uporządkowany zbiór elementów), a nie tylko skalar. Wymiar w(E) wyrażenia E jest definiowany rekurencyjnie:

```

W (<wyrażenie proste>) = 1
W (<wyrażenie 1>, ..., <wyrażenie k>) =
  w (<wyrażenie 1>) + ... + (wyrażenie k)
w (<wyrażenie blokowe>) =
  w (let <definicja> in <wyrażenie> end) =
  w (<wyrażenie>)

```

```

w (<wyrażenie warunkowe>) =
  w (if<wyrażenie> then<wyr 1> else<wyr 2> end =
  w (<wyr 1>) = w (<wyr 2>)
w (<wywołanie procedury>) =
  w (<nazwa>(<wyrażenie>) =
    <liczba elementów na <liście param wy>
    procedury <nazwa>

```

W <def procedury> wymiar <wyrażenia> w treści procedury musi być równy wymiarowi <listy param wy>.

Często jest wygodne wprowadzanie nazwy dla wyrażeń, które występują w kilku miejscach pewnego większego wyrażenia. Służy temu wyrażenie blokowe

```

let {<deklaracja typu>;
    <lista nazw 1>=<wyr 1>,
    .....
    <lista nazw k>=<wyr k>;
in  <wyrażenie> end

```

Pozwala ono na lokalne użycie nazw, które mogą występować w <wyr 1>... ..<wyr k>, <wyrażeniu>, oraz na wprowadzeniu dodatkowych nazw dla <wyr 1>, ..., <wyr k> stanowiących fragmenty <wyrażenia>. Wymaga się, aby liczba elementów na liście nazw i była równa wymiarowi <wyr i>. Wszystkie wprowadzane nazwy muszą być różne. Wyrażenia blokowe mogą być zagnieżdżane; konflikt nazw rozstrzygają zwykłe reguły zakresu odnoszące się do struktury blokowej [22].

Wyrażenie warunkowe

```
if <wyrażenie> then <wyr 1> else <wyr 2> end
```

ma <wyrażenie> boolowskie wymiaru 1, zaś <wyr 1> oraz <wyr 2> muszą mieć ten sam rozmiar, a odpowiadające sobie składowe tych wyrażeń muszą być tego samego typu.

Wywołanie procedury ma postać P (<wyr>), gdzie <wyr> ma wymiar równy liczbie elementów na <liście param we>, a odpowiednie składowe <wyr> muszą być typu deklarowanego w definicji procedury. Komunikacja parametrów pomiędzy program a procedurę odbywa się przez przekazywanie wartości. Wynikiem wykonania procedury jest wyrażenie o wymiarze i typach składowych określonych przez <listę param wy>.

Prosty przykład programu obliczającego silnię:

```

program
  silnia = procedure (n : integer)
    yields integer;

```



```

produkt = procedure (n1:integer,n2:integer)
yields integer;
  if n2 ≤ n1 then n1
  else let m:integer:
    m := (n1 + n2)/2;
    in produkt (n1,m) *
      produkt (m+1,n2)
  end
end
end produkt;
if n < 0 then error
else produkt (1,n) end
end silnia
end

```

8.3. Dynamiczne struktury danych

Język SVAL dysponuje dwoma rodzajami dynamicznych struktur danych: struktury (structure) oraz listy pośredniczące (stream). Pierwszy z tych rodzajów ma zastosowanie uniwersalne - służy do tworzenia przetwarzanych obiektów. Drugi rodzaj jest przeznaczony do organizacji przesyłania informacji pomiędzy modułami (procedurami) programu (por. 2.5). Dynamizm struktur oznacza możliwość ich modyfikacji w trakcie prowadzenia obliczeń.

Strukturę (structure), zgodnie z klasyfikacją Turskiego [22], można określić jako strukturę półregularną. Jest to więc konstrukcja bardzo ogólna, jej szczególnymi przypadkami są na przykład tablica i rekordy. Najprostszą strukturą jest struktura pusta (nil). Ogólnie struktura jest wektorem n-elementowym

$$(s_1 : v_1, s_2 : v_2, \dots, s_n : v_n)$$

gdzie $s_1, s_2, \dots, s_n$ są nazwami (selektorami) składowych struktury, zaś $v_1, v_2, \dots, v_n$ - wartościami tych składowych. Wymaga się, aby selektory składowych struktury były różne. Wartościami składowych struktury mogą być wartości typów prostych wymienionych w punkcie 8.2, lecz także wartościami składowych mogą być inne struktury. Poniżej podano operację, jakie można wykonywać na strukturach. Symbole występujące jako argumenty tych operacji oznaczają: d jest strukturą, s jest selektorem, zaś c pewną wartością.

create() tworzy strukturę pustą nil.

append(d,s,c) tworzy nową strukturę d', która od struktury d różni się tylko tym, że składowa s nowej struktury ma wartość c, niezależnie od tego, czy struktura d taką składową posiada.

delete(d,s) tworzy nową strukturę d, która nie zawiera składowej s, a pozostałe składowe ma identyczne ze składowymi struktury d.

select(d,s) jeżeli struktura d zawiera składową s, to wynikiem operacji jest wartość tego komponentu; w przeciwnym razie wynik operacji jest nieokreślony.

nil-structure(d) jest predykatem przyjmującym wartość prawda, gdy d jest strukturą pustą; w przeciwnym razie predykat przyjmuje wartość fałsz.

Poniżej podano przykład procedury, która działając na dowolnej strukturze binarnej zamienia rolę selektorów l oraz p.

```

odwróć = procedure (x : structure)
yields structure
  if nil-structure(x) then x
  else let lewa, prawa : structure;
    lewa = odwróć(select(x,p));
    prawa = odwróć(select(x,l));
  in append (append
    (create( ),l,lewa),p,prawa)
  end
end
end odwróć

```

Specjalnym rodzajem struktur, przeznaczonym do komunikacji pomiędzy modułami programu są listy pośredniczące (stream). Stanowią one analogon kanałów buforowanych w komunikacji pomiędzy współpracującymi procesami (rozdz. 6). Deklaracja

stream of T

gdzie T jest dowolnym typem, różnym od typu potokowego, definiuje ciąg złożony z obiektów typu T, na którym można wykonywać następujące operacje:

(s,s' oznaczają listy, c wartość typu T):

() daje listę pustą, o zerowej długości.

cons(c,s) tworzy listę s', którego pierwszym elementem jest c, a pozostałe elementy są elementami listy s.

first(s) daje wartość z pierwszego elementu listy; gdy s jest pusta, operacja jest niezdefiniowana.

rest(s) daje w wyniku listę s', która powstaje z s przez opuszczenie pierwszego elementu.

empty(s) jest predykatem prawdziwym, gdy lista s jest pusta i fałszywym w przypadku przeciwnym.

Operacje spełniają spodziewane tożsamości:


```

if empty(s) then s =
else s = cons (first(s), rest(s)) end

```

Osobny przykład wykorzystania list pośredniczących przedstawia procedura generacji liczb pierwszych. Procedura oparta na prezentowanym w rozdziale 6. algorytmie sita Eratosthenesa generuje liczby pierwsze mniejsze od zadanej liczby n . Procedura generuj tworzy ciąg kolejnych liczb całkowitych, poczynając od liczby 2. Wytworzona przez nią lista jest przetwarzana przez procedurę sito, która eliminuje liczby nie będące pierwszymi. Działa ona w taki sposób, że pobiera pierwszy element ze swego wejścia i usuwa wszystkie jego wielokrotności oraz wywołuje rekurencyjnie procedurę sito do pozostałych elementów z pośredniczącej listy wejściowej:

```

liczby pierwsze = procedure (n : integer)
yields stream of integer;
  generuj = procedure (l,n : integer)
    yields stream of integer;
    if l < n then
      else cons (l,generuj(l+1,n) end;
    end generuj;
  sito = procedure (s : stream of integer)
    yields stream of integer;
    if empty (s) then
      else let x : integer;
        s2,s3: stream of integer;
        x,s2 = first(s), rest(s);
        s3 = skreśl (x,s2);
        in cons (x,sito(s3)) end
    end
  end sito;
  skreśl = procedure (x : integer, s : stream of integer)
    yields stream of integer;
    if empty(s) then
      else let y : integer;
        s2,s3 : stream of integer;
        y,s2 = first(s), rest(s);
        s3 = skreśl(x,s2);
        in if divide (x,y) then s3
          else cons (y,s3) end
        end
    end
  end skreśl;

```

```

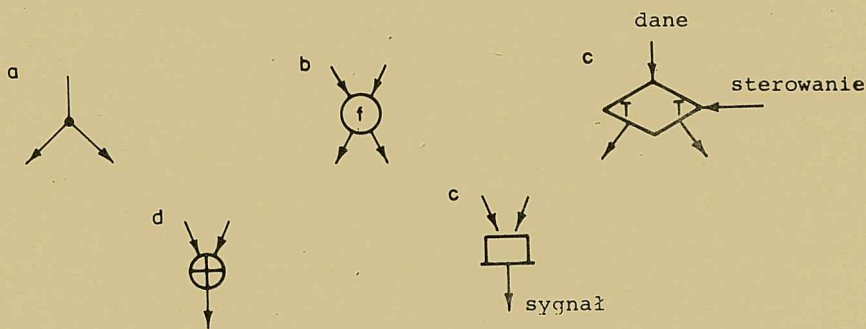
    sito(generuj(2,n));
  end liczbypierwsze;

```

Występujący w procedurze skreśl predykat devide(x,y) przyjmuje wartość prawda, gdy y jest podzielne przez x.

8.4. Schematy przepływu danych

Tradycyjnym sposobem wyrażania semantyki programów sekwencyjnych są schematy blokowe. Przedstawiają one operacje z ich argumentami oraz określają przepływ sterowania (kolejność realizacji operacji). Odpowiednikiem schematów blokowych w programowaniu synchronizowanym przepływem danych są grafy (schematy) przepływu danych.



Rys. 8.1. Graficzna reprezentacja bloków

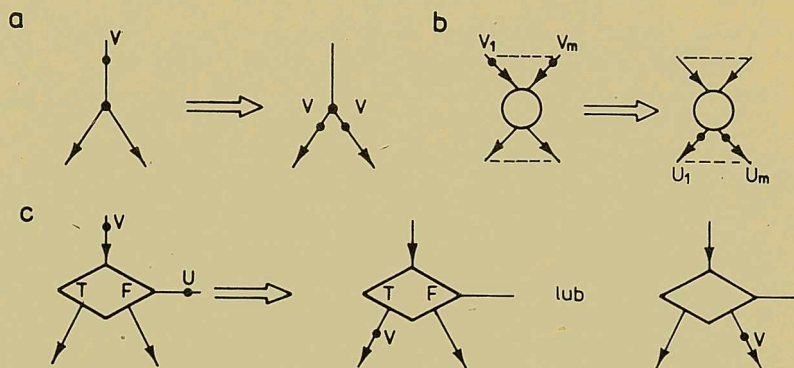
Przez schemat przepływu danych (data flow schema) będziemy rozumieć graf skierowany złożony z wierzchołków zwanych blokami oraz łączących je łuków. Wyróżnia się pięć typów układów - rys. 8.1.:

- (a) replikacyjny,
- (b) funkcyjny
- (c) przełączający,
- (d) scalający,
- (e) pochłaniający.

Schemat przepływu danych jest typu (m,n) jeżeli ma m bloków replikacyjnych nie posiadających łuków wejściowych oraz n bloków replikacyjnych nie posiadających łuków wyjściowych. Bloki te będą nazywane odpowiednio blokami wejściowymi i wyjściowymi schematu. Schemat uważa się za poprawny wówczas, gdy wszystkie pozostałe bloki mają wymagane łuki i wszystkie te łuki mają połączone oba końce z różnymi blokami.

Definicja semantyki operacyjnej schematów przepływu danych wymaga dodatkowych określeń. Konfiguracją schematu nazywa się graf schematu wraz z odcenowaniem pewnych łuków grafu. Odcenowany łuk będzie zaznaczony graficznie przez czarną kropkę (znacznik). Znacznik na łuku symboli-

zuje fakt przekazywania informacji pomiędzy blokami. Rozpływ znaczników reprezentuje przekazywanie informacji pomiędzy współdziałającymi blokami.

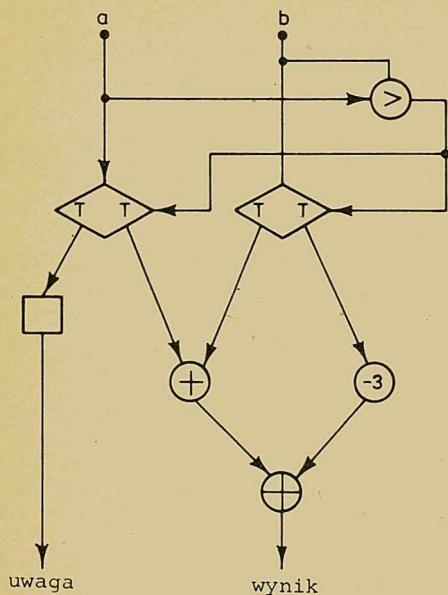


Rys. 8.2. Przykłady reguł wzbudzania bloków

Obliczenia reprezentowane przez (m,n) schemat przepływu danych wyrażają się w postaci ciągu konfiguracji tego schematu: od konfiguracji początkowej, poprzez pośrednie, aż do konfiguracji końcowej. Przejście od jednej konfiguracji do drugiej wynika z zasady zmiany znakowań dla poszczególnych typów bloków (rys. 8.2). Do opisu tych zasad używa się języka stanowiącego modyfikację terminologii sieci Petriego: mówimy więc, że blok jest przygotowany, może się wzbudzić i generować nowe znakowanie.

Koniecznym warunkiem przygotowania bloków jest to, aby ich łuki wyjściowe nie były oznakowane. Blok jest przygotowany - z wyjątkiem bloku scalającego - gdy, dodatkowo, jest oznakowany każdy jego łuk wejściowy. Przygotowany blok aplikacji wzbudza się, pochłania znakowanie łuku wejściowego i powiela je na swych łukach wyjściowych (rys. 8.2a). Podobnie działa blok funkcyjny (8.2b). Wzbudzony blok przełączający generuje znacznik tylko na jednym ze swych wyjść, zależnie od wartości u związanego z nim warunku logicznego (rys. 8.2c). Blok pochłaniający po wzbudzeniu generuje na wyjściu wyróżnioną wartość, zwaną sygnałem, wymaganą do implementacji schematów aplikacji opisanych dalej. Blok scalający działa nieco inaczej od poprzednich bloków. Jeżeli oznakowane jest jedno jego przejście, to wartość znakująca jest bez zmiany przenoszona na wyjście. Natomiast jeżeli oznakowane są oba wejścia, to obie wartości znakujące są przenoszone na wyjście w niezdeterminowanej kolejności.

Tworzenie schematów przepływu przez dowolne łączenie bloków łatwo może prowadzić do powstania sytuacji z blokadami (zastojami) lub niedeterministycznego działania. Dlatego wprowadza się pojęcie klasy tzw.



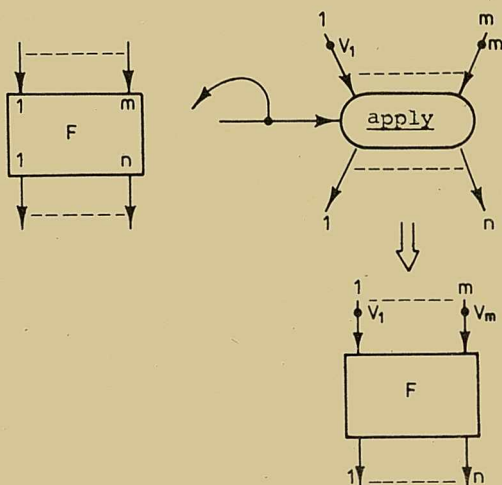
Rys. 8.3. Przykład schematu warunkowego

prawidłowych schematów przepływu danych. Schemat (m,n) jest prawidłowy, gdy graf schematu jest acykliczny i składa się z bloków replikacyjnych, funkcyjnych lub pochłaniających oraz z tzw. podschematów warunkowych. Pojęcie schematu warunkowego wiąże się z wyrażeniem warunkowym; jego sens wyjaśnia przykład schematu z rys.8.3, który oblicza wartość wyrażenia:

if $a > b$ then $a + b$ else $b - 3$ end

Pokazane tam wyjście dodatkowe trig służy wskazaniu, iż obliczenia zostały zakończone.

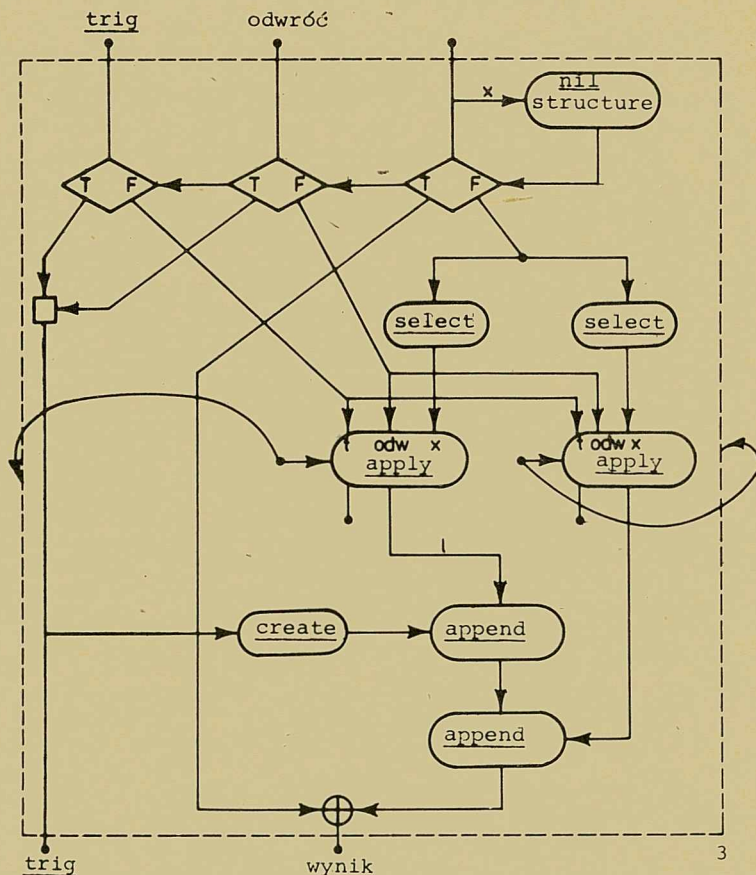
Klasa schematów prawidłowych jest zbyt uboga, aby można za jej pomocą wyrazić takie elementy, jak procedurę, jej wywołanie oraz rekursję. Dlatego wprowadza się dodatkowy typ bloku zwa-



Rys. 8.4. Blok aplikacji

nego blokiem aplikacji (apply) - rys. 8.4. Pierwszym wejściem do bloku aplikacji jest znacznik związany z (m,n) prawidłowym schematem przepływu danych. Blok jest przygotowany wówczas, gdy istnieją znaczniki na wszystkich jego wejściach. Efektem wzbudzenia bloku jest zastąpienie

przez wyspecyfikowany (m,n) schemat. Schemat ten może ponownie zawierać blok aplikacji, umożliwiając tym samym wyrażenie rekursji.



Rys. 8.5. Schemat przepływu danych procedury odwróc

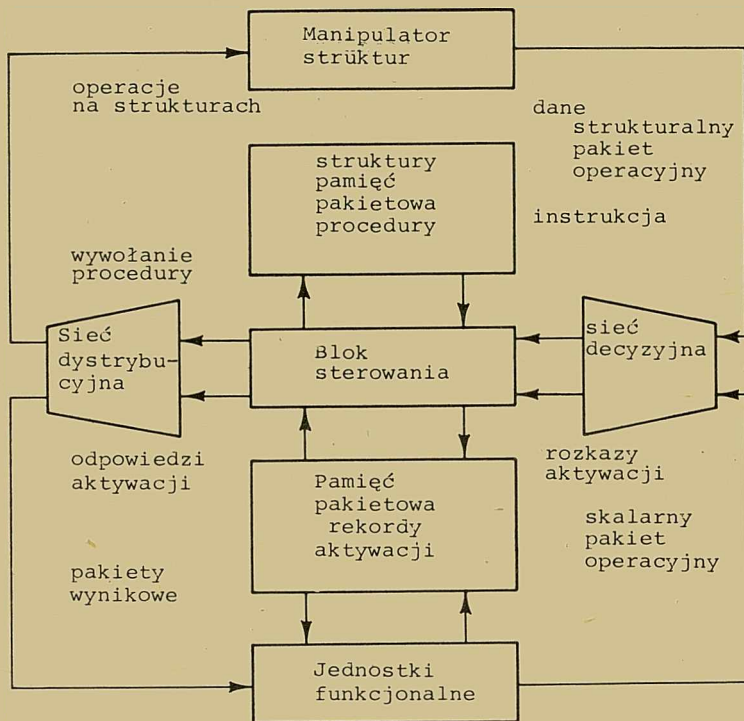
Przykład użycia bloku aplikacji pokazano na rys.8.5, na którym przedstawiono schemat przepływu danych procedury odwróc, zdefiniowane w p.8.3. Występujący tam sygnał trig spełnia rolę synchronizatora schematu: jego obecność na wyjściu jest warunkiem rozpoczęcia obliczeń, zaś jego obecność na wyjściu oznacza zakończenie obliczeń. Sygnał odw jest symbolem wywołania procedury odwróc.

8.5. Problemy implementacji schematów przepływu danych

Efektywna implementacja schematów przepływu danych wymaga nowych, niekonwencjonalnych architektur komputerowych. Przykład koncepcji odpo-

wiedniej architektury podaje [18]. Strukturę komputera przedstawiono na rys. 8.6, na którym wyróżniono sześć podzespołów:

- jednostki funkcjonalne,
- manipulator struktur,
- blok sterowania,
- sieć decyzyjną,
- sieć dystrybucyjną,
- pamięć pakietową.



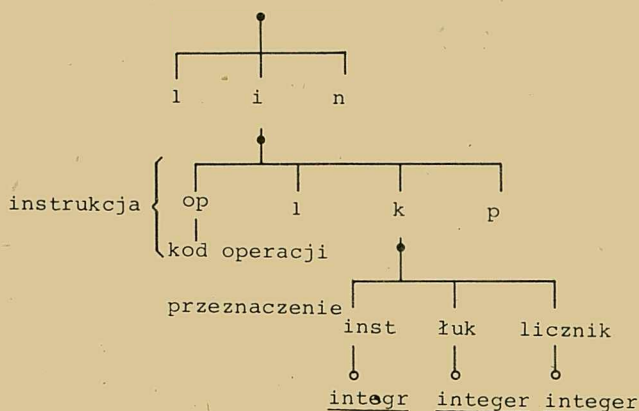
Rys. 8.6. Procesor przepływu danych

Blok sterowania pobiera instrukcje i operandy z pamięci pakietowej, a następnie formuje je w pakiety operacyjne. Każdy taki pakiet przechodzi przez sieć decyzyjną, która transmituje go do odpowiedniej jednostki funkcjonalnej - w przypadku operacji skalarnych, bądź do jednostki manipulacji strukturami - w przypadku działań na strukturach. Wykonanie instrukcji przez jednostki funkcjonalne generuje pakiety wyników, które poprzez sieć dystrybucyjną są przesyłane do bloku sterowania. W bloku sterowania następuje ich połączenie z innymi operandami dla aktywacji dalszych instrukcji.

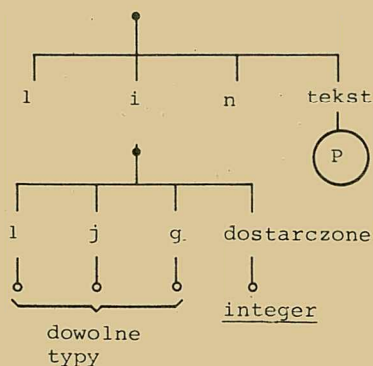
Pamięć pakietowa zawiera zbiór tzw. struktur elementarnych, które są strukturami jednopoziomowymi o składowych przyjmujących wartości skalarne i identyfikowanych przez unikalne nazwy. Zbiór struktur elementarnych tworzy graf cykliczny, w którym każdy łuk jest reprezentowany przez unikalny identyfikator składowej struktury elementarnej stanowiącej wierzchołek wyjściowy łuku. Pamięć pakietowa chroni także przed niepowołanym dostępem do struktur elementarnych oraz zarządza pamięcią fizyczną.

Struktury danych w pamięci pakietowej spełniają potrójną rolę w schematach przepływu danych. (1) jako operandy operacji manipulacji na

a struktura proceduralna



b rekord aktywacji



Rys. 8.7. Struktura proceduralna i aktywacyjna

strukturach, (2) jako procedury, których komponentami są instrukcje procedury, (3) jako rekordy aktywacji, gromadzące wartości operandów dla instrukcji oczekujących na przygotowanie warunków ich realizacji.

Schematy przepływu danych są reprezentowane w procesorze przez tzw. struktury proceduralne (rys. 8.7a). Struktura proceduralna odpowiada schematowi z n blokami ponumerowanymi od 1 do n . Każda składowa nazywana instrukcją, zawiera kod bloku i zbiór jego łuków wejściowych. Każdy łuk wejściowy określa:

- instrukcję, dla której przeznaczony jest wynik,
- łuk wejściowy instrukcji przeznaczenia,
- liczbę operandów wymaganych przez instrukcję przeznaczenia.

Ponieważ w danym momencie może jednocześnie być aktywnych wiele egzemplarzy tego samego schematu przepływu danych, każdy taki egzemplarz jest zaprezentowany przez tzw. rekord aktywacji. Każdy blok jest identyfikowany przez parę (A, I) , gdzie A jest numerem kolejnego rekordu aktywacji, zaś I liczbą przypisaną blokowi w strukturze proceduralnej. Wartość v na k -tym łuku wejściowym bloku (A, I) odpowiada pakietowi wyników, który przenosi informacje $(A, I, k, v, \text{licznik})$, gdzie licznik jest liczbą operandów wymaganą do przygotowania bloku.

Przygotowanie bloku jest wykrywane przez sprawdzenie licznika z liczbą dostarczonych operandów, zarejestrowanych w i -tej składowej rekordu aktywacji. Po przygotowaniu (A, I) bloku, instrukcja bloku zostaje pobrana z I -tej składowej struktury proceduralnej.

Rekord aktywacji (rys. 8.7b) ma składowe 1, ..., n , dla rekordów operandów oraz składową tekst, która jest strukturą proceduralną do uaktywnienia. (W opisywanej implementacji składowa ta jest dzielona dla wszystkich aktywacji tego samego schematu.) Rekord operandów musi mieć tyle podkomponentów, ile zawiera łuków wejściowych, a także podkomponent wskazujący liczbę dostarczonych pakietów wynikowych. Ponieważ rekordy aktywacji zapamiętują wartości przybyłych pakietów wynikowych, operacje na rekordach aktywacji modyfikują jego komponenty. Są to operacje:

- create-activation (P) -

tworzy nowy rekord aktywacji mający P jako komponent tekst, i nie posiadający innych komponentów.

- insert (A, I, k, v) -

dodaje wartość v jako k -ty operand I -tej instrukcji w rekordzie aktywacji A . Dodatkowo zwiększa o 1 składową "dostarczona" tego rekordu.

- remove (A, I) -

usuwa składową I z rekordu A ; jest wykonywana przez blok sterowania, gdy jest generowany pakiet operacyjny dla bloku (A, I) .

- free (A) -

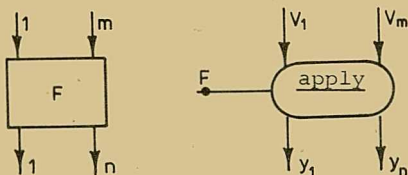
usuwa całkowicie rekord aktywacji A .



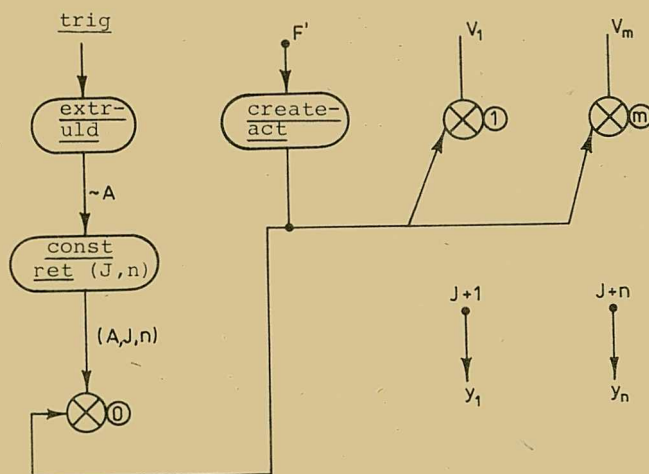
oraz

Rys. 8.8. Notacja pomocnicza

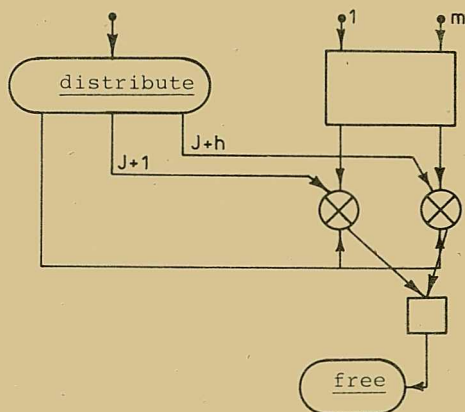
a



b



c



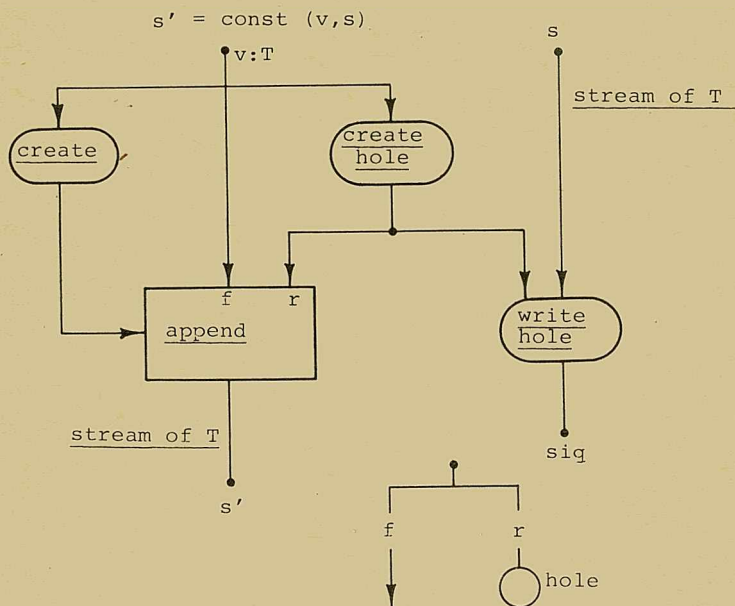
Rys. 8.9. Implementacja bloku aplikacji

Dla każdego przybyłego pakietu wynikowego $(A, I, k, \text{licznik}, v)$ blok sterowania wykonuje operację insert (A, I, k, v) i testuje nową wartość składowej "dostarczone" oraz "licznik". Jeżeli wielkości są równe, instrukcja jest pobierana z pamięci i wykorzystana, wraz z rekordem operandów, do konstrukcji pakietu operacyjnego kierowanego do sieci decyzyjnej. 1-ta komponenta rekordu aktywacji A jest teraz zwalniana.

Implementację bloku aplikacji pokazano na rys. 8.9; używana na nim notacja (rys. 8.8) oznacza insert $(A, i, 1, v)$. Nowe bloki extr-uld, const-ret oraz distribute będą objaśnione dalej. Implementacja zakłada numerację bloków w każdym rekursywnym schemacie zgodnie z zasadami:

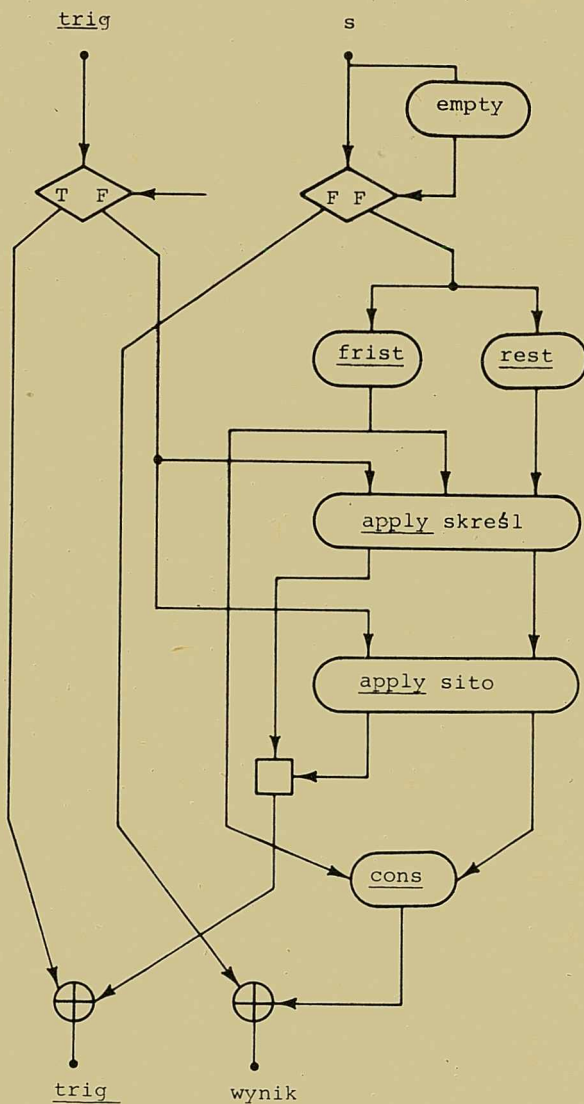
- bloki wejściowe są numerowane $1, \dots, m$;
- bloki replikacji, które otrzymują n wartości wynikowych ze schematu aplikacji są numerowane $J+1, \dots, J+n$, gdzie J jest pewną ustaloną liczbą;
- blok replikacji oznaczony przez o otrzymuje pakiet (A, J, n) zawierający informację wymaganą do konstrukcji pakietu wynikowego po wykonaniu procedury;
- pozostałe bloki mogą być numerowane dowolnie.

Schemat z rys. 8.9 działa następująco: Blok create-act tworzy nowy rekord aktywacji A' , zawierający F' w komponentcie tekst, i przekazuje go



Rys. 8.10. Implementacja operacji const

do bloków insert związanych z wejściami $v_1, \dots, v_m$. Bloki te tworzą pakiety w postaci nowej kopii F' . W tym samym czasie bloki extr-uld oraz const-ref tworzą wartość zwrótną (A, J, n) i przesyłają ją do bloku replikacji O schematu F . Gdy zostaną wyprodukowane wielkości $y_1, \dots, y_n$, bloki distribute oraz insert generują pakiet wynikowy $(A, J+1, 1, 1, y_1)$, który dostarcza wyprodukowane wielkości do wołającego schematu. Blok free



Rys. 8.11. Schemat przepływu danych procedury sito

zwalniania rekord aktywacji, a jego kopia zostaje zwrócona do puli wolnych kopii zarządzanych przez pamięć pakietową.

Implementacja list pośredniczących polega na przedstawieniu ich w postaci pewnych struktur. Lista pośrednicząca jest strukturą, w której składowa f reprezentuje pierwszy element listy, zaś składowa r - pozostałe elementy. Pusta lista jest reprezentowana przez strukturę nil. Operacje na liście s zamienia się na odpowiednie operacje na strukturach: first(s) oraz rest(s) są więc zastąpione przez select (s,f) oraz select (s,r).

Zapewnienie możliwości dopisywania nowych elementów do list i jednocześnie konsumpcję list przez moduły programowe, wymaga takiego rozszerzenia koncepcji struktur danych, aby struktura taka była dostępna jeszcze przed całkowitym jej utworzeniem. Wykorzystuje się w tym celu koncepcję tzw. ramek (holes). Idea tej koncepcji jest urzeczywistniona w implementacji operacji cons opisanej na rys. 8.10.

Create-hole oraz write-hole są specjalnymi operatorami zdefiniowanymi następująco. Blok create-hole pobiera z pamięci pakietowej odpowiedni wolny pakiet. Pakiet ten nazywa się ramką; może ona być niezapełniona lub zapełniona. W stanie niezapełnionym wszystkie operacje na ramkach są kolejkowane, z wyjątkiem operacji write-hole. Po wykonaniu operacji write-hole(H,v) ramka H zmienia swój stan z niezapełnionego na zapełniony i zapamiętuje wartość v . Dotychczas kolejkowane operacje na tej ramce oraz dalsze operacje są już wykonywane bez dalszych opóźnień. Kolejne wykonanie operacji write-hole na ramce H jest nielegalne.

Dla ilustracji równoległości dostarczonej przez taką implementację list rozpatrzmy procedurę sito (rys. 8.11). Zauważmy, że wyjście z pierwszej aktywacji procedury sito będzie strukturą danych zawierającą pierwszy element wynikowej listy pośredniczącej oraz ramkę oczekującą na wypełnienie przez strukturę danych wytworzoną przez rekursywną aktywację sita.

ĆWICZENIA

1. Przedstawić schemat przepływu danych procedury silnia z p. 8.2.
2. Napisać program w języku SVAL sortowania ciągu liczbowego. Przedstawić schemat przepływu danych tego programu.
3. Przedstawić schemat blokowy przepływu danych i napisać w języku SVAL program obliczenia wartości wielomianu.
4. Przedstawić schemat blokowy przepływu danych i napisać w języku SVAL program generacji uporządkowanego ciągu liczb postaci, $2a^3b^5c$ ($a, b, c > 0$) bez powtórzeń i bez pominięć (por. p.5.4).
5. Zmodyfikować treść procedury odwrócić z p. 8.3 w taki sposób, aby mogła działać na dowolnej, a nie tylko binarnej, strukturze.

DODATEK:
MECHANIZMY OBLICZEŃ ASYNCHRONICZNYCH W JĘZYKU PL/I

D.1. Wstęp

Język programowania PL/I powstał w koncernie IBM w pierwszej połowie lat sześćdziesiątych. Zamierzeniem firmy było wprowadzenie języka o wysokim poziomie, który w możliwie szerokim zakresie spełniałby funkcje języka uniwersalnego do tworzenia programowania użytkowego komputerów serii IBM 360. Prace projektowe były prowadzone niemal równolegle z budową systemu IBM 360. Uczestniczyło w nim wielu specjalistów z innych firm, zapoczątkowały one też prace teoretyczne nad zagadnieniem formalnego opisu języków programowania (metoda wiedeńska).

Oceny języka, po wielu latach jego użytkowania, raczej jednoznacznie skłaniają się ku opinii, że PL/I nie spełnił całkowicie oczekiwań, jakie w nim pokładano. Określa się go mianem języka-omnibusu, który stanowi wprawdzie bogatą, ale i nie całkowicie uporządkowaną kolekcję mechanizmów zaczerpniętych od swoich poprzedników - takich jak Fortran, Algol 60 i Cobol. Historia rozwoju języków programowania wskazuje na to, że tego rodzaju zamierzenie - przy ówczesnym stanie teorii - było z góry skazane na niepowodzenie. (Podobne opinie formułuje się zresztą i obecnie, np. w związku z projektem języka Ada.)

Język PL/I jest jednak dość rozpowszechniony w Polsce i krajach RWPG, dzięki rozwijaniu komputerów serii Riad opartych na serii IBM 360. Dlatego mechanizmy języka, które umożliwiają tworzenie programów współbieżnych zasługują na omówienie.

Mechanizmy obliczeń współbieżnych w PL/I bazują na modelu obliczeń w środowisku zwartym. Zasadniczymi przesłankami, które zadecydowały o ich wprowadzeniu było:

- a) stworzenie możliwości pełnego wykorzystania autonomicznie pracujących bloków komputera, takich jak kanały wejścia/wyjścia i równoległe procesory,
- b) stworzenie mechanizmów synchronizacji wykonywania programów współbieżnych.

Bliższe porównanie PL/I z assemblerem, komputerów IBM 360 pozwala stwierdzić, że mechanizmy synchronizacji w obu tych językach są niemal identyczne. Oznacza to, że są to mechanizmy elementarne, niestrukturalizowane.

Ponieważ dodatek spełnia tylko funkcje informacyjne o wybranych mechanizmach PL/I, mające ułatwić podjęcie decyzji o wykorzystaniu języka do obliczeń współbieżnych i zarysować związane z tym problemy, dlatego zmieniono notację języka (np. wprowadzenie podkreśleń słów kluczowych), a także pominięto te problemy, których wyjaśnienie wymagałoby dokładniejszego omówienia konstrukcji sekwencyjnych (np. pominięto zasady dostępu do wspólnych plików).

Ogólny przegląd konstrukcji języka PL/I przedstawia artykuł J. Borowca, Wprowadzenie do języka PL/I, w książce: Problemy przetwarzania informacji t.1, WNT 1970. Pełny opis języka jest zawarty w książce: Opis języka PL/I, WNT 1975 oraz w dokumentacji: PL/I - opis języka, cz. I, cz. II, Centrum Mera-Elwro 1978.

D.2. Zadania i zdarzenia

Statyczna struktura programu w PL/I nie wprowadza jawnie pojęcia procesu jako jednostki strukturalnej. Program ma strukturę blokową, zawierającą definicje obiektów lokalnych i ciągi instrukcji. Procesy, w terminologii PL/I zadania - task, są obiektami dynamicznymi powstającymi podczas wykonywania ciągu instrukcji bloku. Kreacja procesów odbywa się przez wykonanie odpowiednich instrukcji. Nowo utworzone procesy wykonują się współbieżnie z ciągiem instrukcji, który je powołał do życia. Procesy te w trakcie swej realizacji mogą kreować dalsze procesy. Cały program jest więc procesem, którego realizacja rozpoczyna się automatycznie po odpowiedniej akcji systemu operacyjnego nadzorującego wykonanie programu, a dalsze procesy potomne są tworzone dynamicznie.

Kreacja procesu polega, ogólnie mówiąc, na wykonaniu szczególnej postaci instrukcji procedury. Wywołaną procedurą może być procedura zdefiniowana w danym procesie (zadaniu) lub też mogą być pewne procedury standardowe (procedura współpracy z operatorem - DISPLAY, procedury grupowego przesyłania danych - READ, WRITE, DELETE ...). Jedną z form takiego wywołania procedury ma postać:

call P(LP) task(Z),

gdzie P jest nazwą procedury, LP - listą jej parametrów aktualnych, zaś Z jest identyfikatorem stanowiącym nazwę nowo utworzonego procesu (zadania).

Jeżeli w pewnym zdaniu X jest zadeklarowana procedura P i w trakcie jego realizacji zostanie wykonana podana wyżej instrukcja procedury, to nastąpi wykreowanie nowego zadania o nazwie Z, którego treścią będzie treść procedury P zmodyfikowana odpowiednim zastąpieniem listy parametrów formalnych przez parametry aktualne. Zadania Z oraz X będą wykony-

wane współbieżnie i niezależnie w tym sensie, że nie mogą bezpośrednio wpływać na swe zakończenie.

Nadanie nazwy nowo wykreowanemu zadaniu służy jedynie do tego, aby można w innym zadaniu sprawdzać lub zmieniać priorytet tego zadania. Priorytet stanowi atrybut każdego zadania, które służy systemowi operacyjnemu nadzorującemu wykonanie programu, do podejmowania decyzji o pierwszeństwie zadań przy ubieganiu się o przydział pewnych zasobów (czas procesora, urządzenia wejścia/wyjścia). Zadanie Z, które zostało wykreowane w pewnym zadaniu X, zgodnie z przedstawionym wyżej schematem, uzyskuje taki sam priorytet, jak zadanie X. W momencie kreacji zadaniu można przypisać dowolny priorytet; uzyskuje się to przez wywołanie postaci

call P(LP) task (Z) priority (W)

lub

call P(LP) priority (W),

gdzie W jest wyrażeniem całkowitoliczbowym.

Utworzone zadanie uzyskuje priorytet, którego wartość jest równa sumie wyrażenia W i priorytetu zadania, w którym nastąpiło jedno z tych wywołań. Pierwsze z wywołań, nadając nazwę zadaniu, umożliwia też dynamiczną zmianę priorytetu, drugie z wywołań ustala stałą wartość priorytetu bezimiennego zadania.

Współdziałanie zadań (procesów) jest programowane za pomocą mechanizmów synchronizacji opartych na koncepcji zdarzeń. Zdarzenia są tu rozumiane jako fakt osiągnięcia przez sterowanie określonych przez programistę miejsc w procesie. Do rejestracji tego, czy dane zdarzenie zaszło, czy nie zaszło, służą odpowiednie zmienne, zwane zmiennymi zdarzeniowymi. Zmienne zdarzeniowe mają strukturę rekordu

record

completion : boolean;

status : integer

end,

Wartość true na polu completion oznacza, że zdarzenie zaszło, wartość false - że nie zaszło. Wartość pola status ma bardziej bogatą interpretację, tu ograniczymy się do stwierdzenia, że wartość 0 oznacza normalny stan procesu, a każda inna wartość jest kodem pewnego stanu nienormalnego (awaryjnego).

Zmienne zdarzeniowe wprowadza się do programu w postaci jawnej deklaracji lub generuje się je w chwili kreacji zadania. Jawna deklaracja zmiennej zdarzeniowej A ma postać

declare A event;

natomiast dynamiczna generacja może mieć jedną z postaci:

call P(LP) task (Z) priority (W) event (A)

call P(LP) task (Z) event (A)

call P(LP) priority (W) event (A)

call P(LP) event (A).

Zmienna zdarzeniowa deklarowana jawnie jest dostępna do zapisu i odczytu, tak jak każda inna zmienna. Zmienna zdarzeniowa generowana dynamicznie ma ściśle określoną interpretację i jest dostępna tylko do odczytu, natomiast zapis wartości zmiennej jest dokonywany automatycznie w sposób następujący: w chwili kreacji zadania i zmiennej otrzymuje ona wartość false pola completion oraz 0 pola status, podczas wykonywania zadania zmienia się tylko pole status (o ile następuje sytuacja awaryjna), a w momencie zakończenia zadania wartość pola status zmienia się na true. Zasadniczym przeznaczeniem zmiennych zdarzeniowych jest synchronizacja procesów, którą uzyskuje się dzięki wykorzystaniu instrukcji wait postaci

wait (LZ)(W),

gdzie LZ jest listą zmiennych zdarzeniowych, zaś W, występujące opcjonalnie, jest dodatnim wyrażeniem całkowitoliczbowym.

Znaczenie instrukcji jest następujące: wykonanie zadania opóźnia się aż do momentu, gdy zostanie zarejestrowane zajście n zdarzeń przez zmienne zdarzeniowe z listy LZ; n oznacza tu wartość wyrażenia W, brak W powoduje przyporządkowanie domyślne n=1.

Istotnym ograniczeniem przy posługiwaniu się instrukcją wait jest to, że na zajście określonego zdarzenia w danym momencie może oczekiwać co najwyżej jeden proces; oczekiwanie na pewne zdarzenie przez dwa procesy powoduje błąd wykonania. Dodatkowym ograniczeniem przy niektórych kompilatorach języka (kompilator F) jest to, że na zdarzenia wejścia/wyjścia może oczekiwać tylko jeden proces, który zainicjował te operacje.

Opóźnienie procesów realizuje jeszcze jedna instrukcja postaci

delay (W)

gdzie W jest nieujemnym wyrażeniem całkowitoliczbowym. Skutkiem działania tej instrukcji jest zawieszenie zadania na n milisekund, gdzie n jest wartością wyrażenia W, po czym zadanie, o ile ma dostatecznie wysoki priorytet pozwalający mu na przydział zajmowanych poprzednio urządzeń zewnętrznych, jest kontynuowane.

D.3. Dostęp do wspólnych zmiennych

Programy w języku PL/I mają strukturę blokową. Struktura blokowa wprowadza znane zasady zasięgu i dostępności zmiennych [22]. Równoległe zadania, które powstały przez odpowiednie wywołania procedur zadeklarowanych w tym samym bloku mają prawo dostępu do tych wszystkich zmiennych, które są dostępne w tym bloku. A zatem zadania mogą mieć dostęp do wspólnych zmiennych. Poprzednio omówione mechanizmy synchronizacji nie zabezpieczają bezpośrednio przed skutkiem niekontrolowanego dostępu do wspólnych zmiennych. Można natomiast mechanizmy te wykorzystać tak, aby osiągnąć wykluczający dostęp do wspólnych zmiennych.

Ze względu na to, że zmienne w języku PL/I mogą mieć przypisywane tzw. atrybuty klasy pamięci, poza normalnymi regułami dostępu wynikającymi ze struktury blokowej, obowiązują jeszcze reguły dodatkowe. Ich omówienie wymaga krótkiego wyjaśnienia pojęcia atrybutu klasy pamięci.

Każdej zmiennej przyporządkowuje się (jawnie bądź nie) jeden z atrybutów:

static, automatic, controlled, based.

Wszystkim zmiennym statycznym (z atrybutem static) jest przydzielana pamięć w chwili rozpoczęcia programu, a zwalniana w momencie jego zakończenia, niezależnie od stopnia zagnieżdżenia bloków, w których są deklarowane.

Przydział (zwalnianie) pamięci zmiennych automatycznych (z atrybutem automatic) odbywa się w momencie aktywacji (deaktywacji) bloku, w których są zadeklarowane.

Zmiennym kontrolowanym (z atrybutem controlled) przydziela się (zwalnia) pamięć programowo poprzez instrukcję allocate (free). Przydział i zwalnianie odbywa się tu metodą stosu: wykonanie allocate tworzy kolejną generację zmiennej umieszczoną w nowej pozycji na szczycie stosu, zaś wykonanie free usuwa generację zmiennej umieszczoną na szczycie stosu.

Zmiennym wskazywanym (z atrybutem based) pamięć przydziela się i zwalnia programowo instrukcjami allocate i free, przy czym nie używa się tu mechanizmu stosu, a do każdej generacji zmiennej można odwoływać się za pomocą wskaźnika (zmienne te stanowią odpowiednik zmiennych typu wskaźnikowego w języku Pascal).

Zwalnianie pamięci zmiennych kontrolowanych i wskazywanych, którym przydzielono pamięć w pewnym zadaniu, o ile nie wystąpiło przez wykonanie operacji free, dokonuje się automatycznie w chwili zakończenia tego zadania.

Wspomniane reguły dodatkowo sprowadzają się do ogólnego stwierdzenia, aby podczas wykonywania zadań odwołania odnosiły się tylko do ist-

niejących zmiennych. W szczególności oznacza to, że:

- do zmiennych statycznych można odwoływać się w dowolnym zadaniu,
- do zmiennych automatycznych można odwoływać się w każdym zadaniu, wygenerowanym w bloku, w którym zmienne są zadeklarowane,
- do zmiennych kontrolowanych można odwoływać się w tych zadaniach, w których są one znane. Jednakże nie wszystkie przydziały pamięci zmiennym tego rodzaju są znane we wszystkich zadaniach. Zadaniu znane są tylko te przydziały, które miały miejsce przed momentem jego inicjalizacji. Zadanie odpowiedzialne za zwolnienie pamięci zmiennych musi brać pod uwagę to, że przedwczesne zwolnienie pamięci może być przyczyną błędu spowodowanego próbą uzyskania dostępu do tych zmiennych przez inne zadanie,
- do zmiennych wskazywanych odnoszą się zasady podobne jak do zmiennych kontrolowanych.

D.4. Przykłady

Przedstawiono dwa przykłady, stanowiące szkielety programów ilustrujących działanie regionu krytycznego oraz warunkowego regionu krytycznego.

W pierwszym przykładzie założymy, że dwa zadania PR1, PR2 będą korzystały ze wspólnych zmiennych zadeklarowanych w zadaniu PRO.

PRO: procedure;

.....

decl P1,S1,V1,P2,S2,V2 event;

P1.completion:= false;

.....

V2.completion:=false;

P1.status:=0;

.....

V2.status:=0;

}
ustalenie
początkowych
wartości
wszystkich
zmiennych
zdarzeniowych

PR1: procedure;

⋮

P1.completion:=true;

wait (S1);

S1.completion:=false;

⋮

(* sekcja krytyczna x)

⋮

V1.completion:=true;

}
wejście
do sekcji

⋮

end PR1;

}
wyjście


```

PR2: procedure
:
:
P2.completion:=true;
wait (S2);
S2.completion:=false;
:
: (* sekcja krytyczna *)
:
V2.completion:=true;
:
:
end PR2;
call PR1 priority (1);
call PR2 priority (1);
do white true;
:
wait (P1,P2)(1);
it P1.completion then
:
S1.completion:=true;
P1.completion:=false;
:
else
:
S2.completion:=true;
P2.completion:=false
:
end;
wait )V1,V2)(1);
it V1.completion then
:
V1.completion:=false
:
else V2.completion:=false
:
end;
end PRO;

```

wejście
do sekcji

wyjście

Wywołanie procedury PRO w pewnym miejscu programu, postaci

call PRO priority (1)

utworzy zadanie, którego realizacja spowoduje kreację dalszych dwóch zadań współdziałających ze sobą poprzez sekcję krytyczną.

Drugi przykład jest rozbudowaną wersją pierwszego przykładu. Zakłada się bowiem dodatkowo, że procesy mogą zawieszać się wewnątrz sekcji krytycznej, o ile nie są spełnione oczekiwane warunki logiczne. Zawieszony proces zwalnia sekcję krytyczną po czym oczekuje, aż do chwili, gdy oczekiwany warunek stanie się prawdziwy. Wtedy zajmuje sekcję i kontynuuje się od miejsca zawieszenia. Zakłada się, że pierwszy proces, oparty na schemacie działania procedury PW1, czeka na warunek B1, zaś drugi proces, oparty na schemacie PW2, czeka na B2.

PW0: procedura;

```

.....
dcl P1,S1,V1,PB1,SB1,
      P2,S2,V2,PB2,SB2 event
dcl zawieszony_1, zawieszony_2 boolean;
P1.completion:=false;
.....
SB2.completion:=false;
P1.status:=0;
.....
SB2.status:=0;
PW1:procedure;
:
: P1.completion:=true;
: wait (S1);
: S1.completion:=false;
:
:   it not B1 then
:     PB1.completion:=true;
:     wait (SB1);
:     SB1.completion:=false
:   end;
:   V1.completion:=true;
:
: end PW1;
PW2:procedure
:
: P2.completion:=true;
: wait (S2);
: S2.completion:=false;
:
:   of not B2 then
:     PB2.completion:=true,
:     wait (SB2)
:     SB2.completion:=false
:   end;
:
: V2.completion:=true;
:
: end PW2;

```

Ustalenie
początkowych
wartości
wszystkich
zmiennych
zdarzeniowych

wejście
do sekcji
krytycznej

zawieszenie
w sekcji
krytycznej

wyjście

wejście
do sekcji
krytycznej

zawieszenie
w sekcji
krytycznej


```

call PW1 priority (1);
call PW2 priority (1);
zawieszony_1:=false;
zawieszony_2:=false;
do while not (zawieszony_1 and zawieszony_2)
  wait P1.completion then
    P1.completion:=false;
    S1 completion:=true;
    wait (VB1,V1)(1);
    if VB1.completion then
      zawieszony_1:=true;
      VB1.completion:=false;
    else
      V1.completion:=false;
      if zawieszony_2 then
        if B2 then zawieszony 2:=false
          SB2.completion:=true
        end
      end;
    end;
  end;
else
  P2.completion:=false;
  S2.completion:=true
  wait (VB2,V2)(1);
  if VB2.completion then
    zawieszony 2:=true;
    VB2.completion:=false;
  else
    V2.completion:=false;
    if zawieszony_1 then
      if B1 then zawieszony_1:=false;
        SB1.completion:=true
      end;
    end;
  end;
end;
end FWO;

```

Wywołanie procedury

```

call FWO event (blokada)
uruchomi kolejno procesy oparte na działaniu procedur FWO, PW1, PW2. Za-

```

kończenie działania procesu PWO, sygnalizowane zmienną zdarzeniową blokadą, będzie oznaczać, że procesy PW1, PW2 są zawieszone w jednoczesnym oczekiwaniu na warunki B1, B2.

ĆWICZENIA

1. Przeanalizować przykłady z p. D.4 i zmodyfikować je w taki sposób, aby wyeliminować stałe uprzywilejowanie jednego z procesów.
2. Zmodyfikować podane przykłady w taki sposób, aby wyeliminować procesy PRO, PWO pośredniczące odpowiednio we współdziałaniu procesów PR1, PR2 oraz PW1, PW2.
3. Napisać w języku PL/I program symulujący działania czytelników i pisarzy (ćwicz. 1, rozdz. 2).
4. Napisać w języku PL/I program symulujący działanie filozofów (ćwicz. 5, rozdz. 2).
5. Napisać w języku PL/I program generacji liczb pierwszych metodą sita Eratosthenesa.
6. Napisać w języku PL/I program przydzielania zasobów zgodnie z algorytmem bankiera (ćwidz. 7, rozdz. 3).

LITERATURA

- [1] AGERWALA T., Some extended semaphore primitives, *Acta Informatica*, vol. 8, 201-220, 1977.
- [2] ALLAN S.J., OLDEHOEFT, A flow analysis procedure for the translation of high level languages. Technical Report No. 79-2, Dept. of Comp.Sc., Iowa State Univ., Ames 1979.
- [3] ANDREWS G.R., The design of message switching system: an application and evaluation of Modula, *IEEE Trans., on Softw. Eng.*, vol. SE-5, 138-147, 1979.
- [4] BAYER R., GRAHAM R.M., SEEGMULLER (ed-s), *Operating systems*, Lect. Notes in Comp. Sc. vol. 60, Springer-Verlag 1978.
- [5] BIAŁASIEWICZ J., ADEREK A., MALISZEWSKI K., *Oprogramowanie podstawowe komputerowych systemów sterowania*, WNT 1979.
- [6] BRINCH HANSEN P., The nucleus of a multiprogramming system, *Comm. of ACM*, vol. 13, 238-241, 1970.
- [7] BRINCH HANSEN P., Structured multiprogramming, *Comm. of ACM*, vol. 15, 574-578, 1972.
- [8] BRINCH HANSEN P., *Podstawy systemów operacyjnych*, WNT 1979 (tłum. z oryginału and. 1973).
- [9] Brinch Hansen P., The programming language "Concurrent Pascal", *IEEE Trans. on Softw. Eng.*, vol. SE1-2, 199-207, 1975.
- [10] BRINCH HANSEN P., The architecture of concurrent programs, Prentice-Hall, 1977.
- [11] BRINCH HANSEN P., Distributed processes a concurrent programming concept, *Comm. of ACM*, vol. 21, 934-991, 1978.
- [12] BRYANT R.E., DENNIS J.B., *Concurrent programming*, MIT/LCS/TM-115, Cambridge 1978.
- [13] BROCK J.D., MONTZ L.B., Translation and optimization of data flow programs, MIT/LCS/TM 181, Cambridge 1979.
- [14] CAMPBELL R.H., HABERMANN A.N., The specification of process synchronization by path expressions, *Lect. Notes in Comp.*, vol. 16, 89-102, Springer-Verlag, 1974.
- [15] COLEMAN D., GALLIMORE R.M., HUGHES J.W., POWELL M.S., An assesement of Concurrent Pascal, *Softw. Pract. and Exp.*, vol. 9, 827-837, 1979.
- [16] DENNIS J.B., First version of data flow procedure language, *Lect. Notes in Comp.Sc.*, vol. 19, 362-376, Springer-Verlag, 1974.
- [17] DENNIS J.B., MISUMAS D.P., A preliminary architecture for a basic data-flow processor, *Proceed. of the Second Annual Symp. on Comp. Architecture*, 126-132, IEEE, 1975.
- [18] DENNIS J.B., WENG K., An abstract implementation for concurrent computations with streams, MIT/LCS/TM 180, Cambridge, 1979.
- [19] DIJKSTRA E.W., The structura of the THE multiprogramming system, *Comm. of ACM*, vol. 11, 341-346, 1968.
- [20] DIJKSTRA E.W., Co-operating sequential processes, w: *Programming Languages*, ed. GENUYS F., Academic Press, N.Y., 1968.
- [21] DIJKSTRA E.W., Hierarchical ordering of sequential processes, *Acta Informatica*, vol. 1, 115-138, 1971.
- [22] DUBIELEWICZ I., HUZAR Z., *Język i programowanie*, Skrypt, Politechnika Wrocławska, 1981.
- [23] ENSLOW P.H.Jr., (ed.), *Systemy cyfrowe wieloprocesorowe*, WNT 1978 (tłum. z ang. 1974).
- [24] FELDMAN J.A., High level programming for distributed computing, *Comm. of ACM*, vol. 22, 353-368, 1979.

- [25] GREIF I., Semantics of communicating **parallel processes**, MIT/MAC/TR-154, Cambridge 1975.
- [26] GRIES D. (ed.), Programming methodology. A collection of articles by members of IFIP WG2.3, Springer-Verlag 1978.
- [27] HARTMANN A.C., A Concurrent Pascal compiler for mini-computers, Springer-Verlag 1977.
- [28] HEWITT C., BAKER H., Laws for communicating parallel processes, Information Processing 77, IFIP, 987-992, North Holland Publ. Comp., 1977.
- [29] HEWITT C., ATKINSON R.R., Specification and proof techniques for **serializers**, IEEE Trans. on Softw. Eng., vol. SE-5, 10-23, 1979.
- [30] HEWITT C., ATTARDI G., LIEBERMANN, Specifying and proving properties of guardians for distributed systems, Lect. Notes in Comp.Sc., vol. 70, 316-336, Springer-Verlag 1979.
- [31] HIRSCHBERG D.S., Fast parallel sorting algorithms, Comm. of ACM, vol. 21, 657-661, 1978.
- [32] HIRSCHBERG D.S., CHANDRA A.K., SARWATE D.V., Computing connected components on parallel computers, Comm. of ACM, vol. 22, 461-464, 1979.
- [33] HOARE C.A.R., Towards theory of parallel programming, w: Operating systems techniques, ed. HOARE C.A.R., Academic Press, 1972.
- [34] HOARE C.A.R., Monitors: an operating systems structuring, Comm. of ACM, vol. 17, 549-557, 1974.
- [35] HOARE C.A.R., Communicating sequential processes, Comm. of ACM, vol. 21, 666-677, 1978.
- [36] HOLDEN J., WAND I.C., An assesement of Modula, Softw. Pract. and Exp., vol. 10, 593-621, 1980.
- [37] HOLT R.G., GRAHAM G.S., LAZOWSKA E.D., SCOTT M.A., Structured concurrent programming with operating systems applications, Addison-Wesley Publ. Comp., 1978.
- [38] HUZAR Z., Uogólnione monitory w języku programowania "Concurrent Pascal", Podstawy Sterowania, t. 10, 43-56, 1980.
- [39] HUZAR Z., Semantyka programów z operacjami wejścia-wyjścia, Podstawy Sterowania, t. 11, 33-50, 1981.
- [40] ICHBIAH J.D., HELIARD J.C., ROUBINE O., BARNES J.G.P., KRIEG-BRUEKNER B., WICHMANN B.A., Preliminary Ada reference manual (Part A), Rationale for design of the Ada programming language (Part B), SIGPLAN Notices, vol. 14(6), 1979.
- [41] KAHN G., Semantics of simple language for parallel programming, Information Processing 74, IFIP, 471-475, North Holland Publ.Comp., 1974.
- [42] KAHN G., MAC-QUEEN D., Coroutines and networks of parallel processes, Information Processing 77, IFIP, 993-998, North Holland Publ. Comp., 1977.
- [43] KAHN G. (ed.), Semantics of concurrent computations, Lect. Notes in Comp. Sc., vol. 70, 1979.
- [44] KELLER R.M., Sentinels: a concept for multiprocess coordination, Dept. of Comp. Sc., Univ. of Utah, UUCS-78-104, 1978.
- [45] KOTOV V.E., O paralelnych jazykach, Kibernetika, No.3, 1-12, No.4, 1-10, 1980.
- [46] KUCK J.D., MURAOKA Y., CHEN S.-C., On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup, IEEE Trans. on Comp., vol. C-21, 1293-1310, 1972.
- [47] KWIATKOWSKI J. Metoda konwersji programu szeregowego na równoległy. Rozprawa doktorska, Inst. Cyber. Techn., Politechnika Wrocławska, Rap. PRE 108, 1980.
- [48] LISKOV B., Primitives for distributed computing, MIT/LSC/TM, Cambridge 1979.
- [49] MC-CRAW J.R., ANDREWS G.R., Access control in parallel programs, IEEE Trans. on Softw. Eng., vol. SE-5, 1-8, 1979.
- [50] MC-KEAG R.M., MILLIGAN P., An experiment in parallel program design, Softw. Pract. and Exp., vol. 10, 637-696, 1980.

- [51] MUNRO J., PATERSON M., Optimal algorithms for parallel polynomial evaluation, *Journal of Comp. and Syst. Sc.*, vol. 7, 189-198, 1973.
- [52] NARAYANA K.T., PRASAD V.R., JOSEPH M., Some aspects of concurrent programming in CCNPascal, *Softw. Pract. and Exp.*, vol. 9, 749-770, 1979.
- [53] NOWICKI Z.M., Organizacja macierzowych jednostek przetwarzania potokowego realizujących operacje matematyczne, *Inst. Podst. Inf.*, PAN, PWN 1980.
- [54] PYLE I.C., Input/output in high level programming languages, *Softw. Pract. and Exp.*, vol. 9, 907-914, 1979.
- [55] RAMAMOORTHY C.V., PARK J.H., LI H.F., Compilation techniques for recognition of parallel processable task in arithmetic expressions, *IEEE Trans. on Comp.*, vol. C-22, 986-998, 1973.
- [56] REED D.P., KANODIA R.K., Synchronization with eventcounts and sequencers, *Comm. of ACM*, vol. 22, 115-123, 1979.
- [57] ROBERT R., VERJUS J.-P., Towards autnomons description of synchronization modules, *Information Processing 77*, IFIP, 981-986, North Holland Publ. Comp., 1977.
- [58] ROBERTS E.S., CLARKE E.M., EVANS A., MORGAN C.R., Task management in Ada: a critical evaluation for real-time multiprocessors, *Aiken Comp. Lab., Harvard Univ.*, TR-07-80, Cambridge 1980.
- [59] SALWICKI A. i inni, Uniwersalny język programowania Loglan 77, *Inst. Inf.*, Uniw. Warszawski, Opracowanie wewnętrzne 1977.
- [60] SCHULTZ H.A., On the design of a language for programming real-time concurrent processes. *IEEE Trans. on Softw. Eng.*, vol. SE-5, 248-255, 1979.
- [61] SHARP J.A., Data oriented program design, *Dept. of Comp. Sc., Univ. College of Swansea*, Internal report, 1980.
- [62] TURSKI W.M., Metodologia programowania, WNT 1978.
- [63] WANG P.-S., LIN M.T., Parallel processing of high level language programs, *Proceed. of the 1979 Intern. Conf. on Parallel Processing*, 17-25, IEEE, 1979.
- [64] WELSH J., BUSTARD D.W., Pascal-Plus: another language for modular multiprogramming, *Softw. Pract. and Exp.*, vol. 9, 947-957, 1979.
- [65] WELSH J., LISTER A., A comparative study of task communication in Ada, *Softw. Pract. and Exp.*, vol. 11, 257-290, 1981.
- [66] WETHERELL C., Design considerations for array processig languages, *Softw. Pract. and Exp.*, vol. 10, 265-271, 1980.
- [67] WIRTH N., Modula: a language for modular multiprogramming, *Softw. Pract. and Exp.*, vol. 7, 3-35, 1977.
- [68] WIRTH N., The use of Modula, *Softw. Pract. and Exp.*, vol. 7, 37-65, 1977.
- [69] WIRTH N., Design and implementation of Modula, *Softw. Pract. and Exp.*, vol. 7, 67-84, 1977.
- [70] WAND I.C., Systems implementation languages and Ironman, *Softw. Pract. and Exp.*, vol. 9, 853-878, 1979.
- [71] Z 200 Recommendation. CCITT High Level Language (Chill), CCITT, Geneva 1981.

SPIS RZECZY

PRZEDMOWA	3
1. WPROWADZENIE	5
1.1. Pojęcia podstawowe	5
1.2. Kierunki rozwoju	7
1.3. Źródła równoległości	10
ĆWICZENIA	14
2. PORÓWNIANIE PODSTAWOWYCH KONCEPCJI	16
2.1. Zasady porównania	16
2.2. Przykładowe systemy	17
2.3. Procesy w środowisku zwartym	18
2.3.1. Semaforzy	18
2.3.2. Monitory	21
2.4. Procesy w środowisku rozproszonym	25
2.5. Obliczenia równoległe synchronizowane przepływem danych ..	31
2.6. Podsumowanie	35
ĆWICZENIA	36
3. JĘZYK PROGRAMOWANIA CONCURRENT PASCAL	39
3.1. Uwagi wstępne	39
3.2. Symbole podstawowe	40
3.3. Struktury danych	40
3.4. Instrukcje	41
3.5. Procesy	43
3.6. Monitory	45
3.7. Klasy	48
3.8. Struktura programu	49
3.9. Wybrane własności implementacji języka na minikomputerach PDP11	50
3.10. Program przykładowy	53
3.11. Uwagi końcowe	60
ĆWICZENIA	61
4. JĘZYK PROGRAMOWANIA MODULA	63
4.1. Wprowadzenie	63
4.2. Mechanizmy sekwencyjne	64

4.3.	Mechanizmy równoległe	68
4.4.	Własności implementacji na minikomputerach PDP11	70
4.5.	Program przykładowy	73
4.6.	Ocena języka	77
	ĆWICZENIA	78
5.	PROCESY W ŚRODOWISKU ROZPROSZONYM	80
5.1.	Wprowadzenie	80
5.2.	Komunikujące się procesy sekwencyjne Hoare'a	81
5.3.	Procesy rozproszone Brinch Hansena	87
5.4.	Sieci procesów Kahna	91
	ĆWICZENIA	98
6.	MECHANIZMY RÓWNOLEGŁOŚCI W JĘZYKU PROGRAMOWANIA ADA	99
6.1.	Wprowadzenie	99
6.2.	Struktura programów	99
6.3.	Komunikacja pomiędzy zadaniami	103
6.4.	Niedeterminizm	104
6.5.	Przykładowe programy	107
	ĆWICZENIA	110
7.	MECHANIZMY RÓWNOLEGŁOŚCI W JĘZYKU PROGRAMOWANIA CHILL	111
7.1.	Wprowadzenie	111
7.2.	Struktura programów	112
7.3.	Komunikacja we wspólnym środowisku	113
7.4.	Komunikacja z wykorzystaniem sygnałów	116
7.5.	Komunikacja z wykorzystaniem buforów	120
	ĆWICZENIA	125
8.	OBLICZENIA SYNCHRONIZOWANE PRZEPŁYWEM DANYCH	126
8.1.	Wprowadzenie	126
8.2.	Koncepcja języka programowania	128
8.3.	Dynamiczne struktury danych	130
8.4.	Schematy przepływu danych	133
8.5.	Problemy implementacji schematów przepływu danych	136
	ĆWICZENIA	143
DODATEK.	MECHANIZMY OBLICZEŃ ASYNCHRONICZNYCH W JĘZYKU PL/I	144
D.1.	Wstęp	144
D.2.	Zadania i zdarzenia	145
D.3.	Dostęp do wspólnych zmiennych	148
D.4.	Przykłady	149
	ĆWICZENIA	153
	LITERATURA	154



242932 W/3

Cena zł 95,—

Skrypty Politechniki Wrocławskiej

są do nabycia w:

P.P. „Dom Książki”

Księgarni Wr 49

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

oraz

Wojewódzkiej Księgarni Technicznej

ul. Świdnicka 8, 50-067 Wrocław