Wydział Matematyki, Informatyki i Mechaniki
Uniwersytetu Warszawskiego
Instytut Informatyki

REALIZACJA PROCESÓW ROZPROSZONYCH W LOGLANIE 82

Bolesław Ciesielski

numer albumu 113269

Praca magisterska wykonana pod kierunkiem dr Marka Lao

Warszawa, 1988

Spis treści

ROZDZIAŁ I

Wstęp

1. Systemy rozproszone i ich znaczenie

Systemy komputerowe pozwalające na wykonywanie programów współbieżnych oraz języki programowania takich systemów już od dłuższego czasu skupiają uwagę ludzi związanych w jakiś sposób z wykorzystaniem komputerów. Szczególnie dużego znaczenia nabrały systemy rozproszone, a złożyło się na to wiele powodów.

Rozwój sieci komputerowych stworzył potrzebę istnienia odpowiednich systemów operacyjnych. Muszą one działać często na różnych komputerach znacznie od siebie oddalonych. Klasycznym przykładem są komputerowe systemy rezerwacji oparte o rozproszone bazy danych. Realizacja takich systemów byłaby niemożliwa lub co najmniej ekonomicznie nieuzasadniona przy założeniu centralizacji zarządzania.

Postępy w dziedzinie techniki mikroprocesorowej i obwodów scalonych wielkiej skali integracji umożliwiły produkcję komputerów, które wykorzystywane przez jednego użytkownika, dysponują jednocześnie mocą obliczeniową wystarczającą do rozwiązania wielu drobniejszych zadań wymagających dotychczas dzielonego dostępu do dużego komputera. Jednak podczas gdy ceny układów elektronicznych systematycznie maleją, koszt urządzeń peryferyjnych jest nadal bardzo wysoki. Obecnie cena dobrze wyposażonej stacji roboczej (ang. workstation) jest przede wszystkim zależna od kosztu stacji dysków i taśm magnetycznych czy nawet drukarki laserowej. Sytuacja taka doprowadziła do powstania lokalnych sieci komputerowych. Komputery włączone do takiej sieci dzielą między siebie kosztowne urządzenia peryferyjne. Potrzebne są zatem systemy operacyjne pozwalające na efektywne i bezpieczne zarządzanie tymi zasobami.

Spoglądając z bardziej filozoficznego punktu widzenia można zauważyć, że rozwój techniki komputerowej, choć bardzo dynamiczny, nie będzie nieograniczony. Kiedy już osiągniemy stan, w którym zmiana wartości jednego bitu będzie odpowiadać przeskokowi elektronu na sąsiedni poziom energetyczny, wtedy tylko zrównoleglenie działania będzie w stanie przyspieszyć komputer. Oczywiście są to spekulacje, gdyż nowe odkrycia fizyki mogą diametralnie zmienić sytuację. Jednak w przypadku superkomputerów

z ograniczeniami czysto fizycznymi zetknięto się już dość dawno. Możliwość wykonywania obliczeń równoległych daje natomiast jakąś szansę zwiększenia szybkości. Choć nie wyobrażamy sobie dzisiaj komputera z 4M procesorami to wynika to raczej ze słabości wiedzy i narzędzi potrzebnych do tworzenia oprogramowania niż możliwości technicznych. Już teraz przecież tworzone są systemy oparte o koncepcję dużej liczby połączonych procesorów z oddzielną pamięcią [EH87].

Nic dziwnego zatem, że każdy nowoczesny język programowania zawiera odpowiednie konstrukcje pozwalające na pisanie programów równoległych i uruchamianie ich w systemach rozproszonych. Umożliwiają one programiście formułowanie algorytmów w terminach procesów (zadań) wykonujących się równolegle. Procesy te nie mogą działać całkowicie niezależnie, jeśli mają służyć rozwiązaniu jednego problemu. Konieczne jest dostarczenie programiście środków koordynacji działań procesów.

Nie ulega wątpliwości, że programowanie systemów rozproszonych jest jakościowo inne niż programowanie systemów z dzieloną pamięcią. Wpływa na to kilka czynników związanych z samą istotą systemów rozproszonych. Podstawową cechą charakterystyczną systemów rozproszonych są ograniczone możliwości komunikacji między procesorami. Zazwyczaj mogą się one komunikować tylko poprzez przesyłanie komunikatów o ograniczonej pojemności. Zatem wiedza zgromadzona w systemie jest też z konieczności rozproszona. Nieopłacalne staje się utrzymywanie centralnego nadzorcy sprawującego kontrolę nad wszystkimi działaniami systemu, co wynika także z tego, że systemy rozproszone są bardziej podatne na awarie i uszkodzenia (np. fizyczne uszkodzenia łączy) niż tradycyjne systemy komputerowe. Jednocześnie poprzez odpowiednią konstrukcję systemów operacyjnych można osiągnąć większą niezawodność dzięki duplikowaniu danych i kanałów komunikacyjnych.

2. Podstawowe cele pracy

Język programowania Loglan rozwijany w naszym Instytucie zawiera mechanizmy umożliwiające tworzenie i synchronizację procesów równoległych. Jednak żadna z istniejących realizacji Loglanu nie udostępniała ich programiście, co wynikało przede wszystkim z braku odpowiednich środków technicznych tzn. komputerów i systemów operacyjnych. Pojawienie się sieci lokalnej umożliwiło podjęcie próby realizacji konstrukcji równoległych w

Loglanie.

Istniejący w Loglanie mechanizm synchronizacji procesów, oparty o semafory binarne, nie nadawał się jednak do realizacji w systemie rozproszonym. Semafory są z definicji zmiennymi dzielonymi i jako takie mają sens tylko w systemach ze wspólną pamięcią.

Główne cele niniejszej pracy są zatem następujące:

- zaprojektowanie mechanizmu synchronizacji procesów dla Loglanu nadającego się do realizacji w systemie rozproszonym
- realizacja tego mechanizmu przy użyciu istniejącego kompilatora Loglanu 82 i sieci lokalnej komputerów IBM PC

ROZDZIAŁ II

Proponowany mechanizm synchronizacji procesów rozproszonych – obce wołanie procedur.

Obce wołanie procedur jest mechanizmem synchronizacji i komunikacji procesów dostosowanym do realizacji w systemie rozproszonym, tzn. nie opierającym się na zmiennych dzielonych. Podstawowa koncepcja pochodzi od spotkania w Adzie [D83]. W proponowanym rozwiązaniu zdalne wołanie procedury w innym procesie (tzn. wołanie procedury poprzez dostęp kropkowy do obiektu procesu) ma semantykę inną niż opisana w raporcie Loglanu [L87]. W niniejszej pracy mechanizm synchronizacji procesów oparty na zmodyfikowanym zdalnym wywołaniu jest zwany obcym wołaniem procedur.

W obcym wywołaniu procedury uczestniczą zarówno proces wołający, jak i proces wołany. Jest to podstawowa różnica w stosunku do zdalnego wołania procedury opisanego w [L87], gdzie procedura jest wykonywana przez proces wołający bez żadnego zaangażowania ze strony procesu wołanego. Dzięki temu, że w obcym wołaniu procedury biorą udział oba procesy, można stworzyć synchroniczny mechanizm komunikacji procesów oparty o właśnie taki sposób wywoływania procedur.

Dla każdego obiektu procesu definiujemy maskę procedur jako podzbiór wszystkich procedur zadeklarowanych w tym procesie na najwyższym poziomie. Mówimy, że procedura jest odblokowana w procesie, jeśli należy do jego maski procedur. W przeciwnym przypadku jest ona zablokowana.

Obce wywołanie procedury jest inicjowane przez proces wołający za pomocą instrukcji:

call X.p(e,v)

gdzie X jest wskaźnikiem do procesu wołanego, p jego procedurą, natomiast e i v symbolizują parametry wejściowe i wyjściowe. Po przekazaniu parametrów wejściowych proces wołający zawiesza się w oczekiwaniu na zakończenie obcego wywołania.

Procedura jest wykonywana przez proces wołany. Zanim proces wołany będzie mógł rozpocząć wykonywanie procedury, muszą być spełnione pewne warunki: proces nie może być zawieszony w żaden sposób (wyjątek – instrukcja **accept** opisana poniżej), a wołana procedura musi być odblokowana w procesie wołanym.

W momencie gdy oba powyższe warunki zostaną spełnione proces wołany zostaje przerwany i zaczyna wykonywać wołaną procedurę (z parametrami przekazanymi przez proces wołający). Przy wejściu do procedury wszystkie procedury w procesie wołanym zostają zablokowane.

Po zakończeniu procedury maska procedur zostaje odtworzona do stanu sprzed wywołania i proces wołany zostaje wznowiony w punkcie przerwania. Proces wołający odczytuje parametry wyjściowe i wznawia działanie po instrukcji wołania.

Ten podstawowy mechanizm jest uzupełniony o instrukcje pozwalające zmieniać maskę procedur procesu. Instrukcje:

enable P1, ..., Pn

oraz

disable P1, ..., Pn

powodują odpowiednio odblokowanie lub zablokowanie procedur o identyfikatorach P1, ..., Pn (w bieżącym procesie). Specjalna forma instrukcji **return**:

return enable P1, ..., Pn **disable** Q1, ..., Qn

pozwala zmienić maskę procedur po tym jak zostanie ona odtworzona po zakończeniu obcego wołania procedury.

Opisany powyżej mechanizm realizuje w pewnym sensie koncepcje przerwań sprzętowych, których obsługa korzysta ze stosu. Instrukcje **enable** i **disable** odpowiadają instrukcjom maszynowym zmieniającym aktualną maskę przerwań, zawartą na przykład w słowie stanu procesora. Instrukcja **return enable** ... **disable** ... jest połączeniem instrukcji zmieniającej poprzednią maskę przerwań zapamiętaną na stosie i instrukcji powrotu z obsługi przerwania.

Dodatkowo przewidziana jest instrukcja:

accept P1, ..., Pn ,

która dodaje do maski (odblokowuje) procedury P1, ..., Pn oraz zawiesza proces w oczekiwaniu na obce wywołanie którejś z aktualnie odblokowanych procedur (być może są wśród nich inne niż P1, ..., Pn). Po zakończeniu wywołanej procedury maska procedur jest odtwarzana do stanu sprzed instrukcji **accept**.

Dzięki instrukcji **accept** można uniknąć aktywnego czekania na obce wywołanie jednej z kilku procedur. Można też zapomnieć o masce procedur i traktować czyste spotkanie (takie jak w BNR Pascalu [G87]) jako szczególny przypadek obcego wołania procedury. Wynika to z tego, że jeśli nie używamy instrukcji **enable/disable** to proces wołany może obsłużyć obce wywołanie procedury jedynie

podczas wykonywania instrukcji **accept**. W innych momentach wszystkie procedury są zablokowane.

Powyższy opis nie jest pełny. Jego celem jest jedynie ogólne przedstawienie istoty zrealizowanego mechanizmu. Szczegółowy opis wraz z ograniczeniami jest zawarty w załączonym podręczniku użytkownika systemu Loglan 82 zrealizowanego na komputerze IBM PC.

ROZDZIAŁ III

Porównanie proponowanego mechanizmu z innymi konstrukcjami występującymi w językach programowania systemów rozproszonych.

1. Sposób ustalania konfiguracji systemu

Ten aspekt języka jest właściwie niezależny od mechanizmu synchronizacji. Chodzi tu o sposób, w jaki programista ustala procesy, z których będzie składał się program. Czasami programista może też określić, w którym momencie procesy będą tworzone i usuwane oraz jakie będą zależności między nimi.

W najprostszym przypadku mamy statycznie ustalony zbiór procesów, które istnieją przez cały czas działania programu. Bardziej rozbudowane języki dają możliwość dynamicznego tworzenia i usuwania procesów w czasie działania programu. Proces, który utworzył dany proces jest nazywany jego ojcem. Zbiór procesów można traktować wtedy jako drzewo, choć nie zawsze zależność ojciec-syn jest pamiętana przez system i ma wpływ na działanie programu.

Rozwiązanie przyjęte przeze mnie było częściowo zdeterminowane istniejącymi konstrukcjami równoległymi w Loglanie. Większość operacji na procesach nie związanych bezpośrednio z mechanizmem komunikacji została zachowana z semantyką prawie nie zmienioną. Do operacji tych należą tworzenie, wznawianie, zatrzymywanie i usuwanie procesów. Oczywiście żadna z tych konstrukcji nie była zrealizowana w istniejącym systemie Loglanu 82, więc właściwie można było narzucić użytkownikowi na przykład wymóg statycznego ustalenia konfiguracji systemu. Być może ułatwiłoby to realizację. Zdecydowałem jednak, że nie należy zmieniać języka w stopniu większym niż to konieczne, szczególnie w sposób niezgodny z charakterem pozostałych konstrukcji. Wydaje się, że wysoka dynamika konstrukcji języka i brak niepotrzebnych ograniczeń były jednymi z celów, jakimi kierowali się twórcy Loglanu.

Jedyną nie zrealizowaną konstrukcją równoległą występującą w Loglanie 82 (nie związaną z mechanizmem synchronizacji) jest wyrażenie **wait**. Jego obliczanie zawiesza proces w oczekiwaniu na zakończenie któregoś z procesów przez niego utworzonych (synów). Wartością wyrażenia jest wskaźnik do tego procesu (lub **none** jeśli proces nie ma nie zakończonych synów). Ta konstrukcja nie została zrealizowana, gdyż w opisywanym systemie poprzednik dynamiczny dla

procesu (ang. dynamic link) jest zapamiętywany tylko do momentu wykonania pierwszej instrukcji **return** (ze względu na sposób wyszukiwania modułu obsługi wyjątków w istniejącym systemie wykonawczym). Brak wyrażenia **wait** jest oczywiście pewną niewygodą, ale daje się ono wydefiniować za pomocą zrealizowanych konstrukcji.

1.1. Konfiguracja statyczna

W wielu prostszych językach występuje tendencja do utożsamienia procesu i procesora. Zbiór procesów ustala się statycznie, w trakcie kompilacji programu. Program równoległy składa się z ciągu deklaracji procesów:

```
process P1;
...
end P1;
...
process Pn;
...
end Pn;
```

gdzie procesy mają własne deklaracje i treść (odpowiadają więc programom w językach sekwencyjnych). Niekiedy możliwe jest utworzenie całej rodziny procesów (o ustalonej liczebności) z tą samą treścią. Wszystkie procesy są tworzone i uruchamiane jednocześnie w momencie rozpoczęcia wykonywania programu. Procesy zostają usunięte i program kończy się w chwili, gdy wszystkie procesy zakończą działanie.

Podejście takie reprezentuje przede wszystkim Brinch Hansen w swoich projektach Concurrent Pascal [BH75] i Distributed Processes [BH78]. Podobna koncepcja występuje też w Moduli [W77]. Języki te służą przede wszystkim do programowania niższych warstw systemów operacyjnych, gdzie ważniejsze jest pozostawienie wielu decyzji programiście niż zapewnienie mu maksymalnej wygody. W porównaniu z proponowanym mechanizmem rozwiązanie takie jest znacznie mniej wygodne i elastyczne. Natomiast weryfikacja programu ze statycznie ustalonym zbiorem procesów jest łatwiejsza niż w przypadku konfiguracji ustalanej dynamicznie. Oczywiście weryfikacja programów równoległych zarówno formalna, jak i nieformalna (np. poprzez testowanie) jest praktycznie bardzo trudna, wręcz na pograniczu niemożliwości. Tym bardziej każde ułatwienie jest istotne.

Realizacja języka wymagającego statycznego ustalenia konfiguracji procesów jest prostsza, co jest efektem niskopoziomowego podejścia. Bardzo upraszcza się zarządzanie pamięcią, szczególnie jeśli w takich językach nie występują procedury rekurencyjne (zdarza się to dosyć często). Wtedy po prostu wystarczy ściśle statyczny przydział pamięci. Także zarządzanie procesorem jest prostsze, ze względu na stałą liczbę procesów.

1.2. Konfiguracja hierarchiczna

W tym przypadku liczba procesów nie jest ustalona w momencie rozpoczęcia wykonywania programu. Tworzenie i usuwanie procesów odbywa się za pomocą instrukcji równoległej typu **cobegin** ... **coend**. Instrukcje **cobegin** mogą być zagnieżdżane, np.:

```
cobegin
  I1;
  I2;
  cobegin
    begin
      I3;
      I4;
      cobegin
        I5;
        I6
      coend
    end;
    I7
  coend;
  I8
coend
```

gdzie I1, ..., I8 oznaczają dowolne instrukcje. Wykonanie instrukcji **cobegin** polega na równoległym wykonaniu instrukcji składowych. Zakończenie wykonywania instrukcji **cobegin** następuje z chwilą zakończenia wykonywania wszystkich instrukcji składowych. W połączeniu z procedurami rekurencyjnymi instrukcja rozwidlenia pozwala na utworzenie nieograniczonej z góry liczby procesów.

W przypadku systemów rozproszonych możliwość dynamicznego rozwidlania procesu istnieje zazwyczaj tylko w obrębie jednego węzła (tzn. zbioru procesorów dzielących pamięć). Wtedy możliwe jest także użycie zmiennych dzielonych. Są to więc języki w pewnym

sensie hybrydowe. Nie zawsze są one projektowane z myślą o systemach rozproszonych. O możliwości (ewentualnie sensowności) realizacji takich języków w systemie rozproszonym decyduje rodzaj występujących w nich konstrukcji (przede wszystkim mechanizmu synchronizacji).

Podejście hierarchiczne reprezentowane jest przez języki Estelle [BD87], Joyce [BH87] i CSP [H78]. Cechą charakterystyczną tego mechanizmu jest fakt, że proces-ojciec nie może się skończyć wcześniej niż proces-syn. Podobny efekt można osiągnąć także w Adzie poprzez deklarację zadań w procedurze. Wywołanie procedury odpowiada wtedy instrukcji **cobegin**.

Mechanizm wprowadzony do Loglanu jest znacznie bardziej dynamiczny. Pozwala on na definiowanie typu procesowego i tworzenie w trakcie wykonywania programu obiektów procesów, która mogą działać właściwie niezależnie od procesu ojca.

1.3. Konfiguracja ustalana dynamicznie

W językach wyższego poziomu istnieje możliwość dynamicznego tworzenia procesów, niezależnie od struktury statycznej. Programista może zadeklarować typ procesowy i zmienne tego typu lub typu wskaźnikowego np.:

```
type
  proc = process(n, k : integer);
         (* deklaracje zmiennych lokalnych i treść *)
         end;
var
  X : proc;
  P : ^proc;
begin
  P := new proc;
  ...
end
```

W powyższym przykładzie proces będący wartością zmiennej X jest tworzony i uruchamiany w momencie opracowywania deklaracji zmiennej, natomiast proces wskazywany przez zmienną P jest tworzony i uruchamiany za pomocą specjalnej instrukcji generującej. Konstrukcje tego rodzaju występują w językach takich jak Ada [DS3], CHILL [R84] i Loglan [L87], jak również w wersji Loglanu opisanej w niniejszej pracy. W Loglanie proces nie może być statycznie zadeklarowany (jak zmienna X powyżej). Natomiast

jeszcze wyraźniej uwidacznia się dynamika języka, bowiem utworzenie obiektu procesu jest oddzielone od jego uruchomienia. Daje to dodatkową elastyczność w połączeniu z prefiksowaniem, gdyż można uruchamiać proces o nie znanym dokładnie typie i parametrach, który został przekazany jako wygenerowany już obiekt.

Proces może być składową innej struktury, np. tablicy procesów czy rekordu. Konstrukcje takie są bardzo wygodne, szczególnie dla programisty tworzącego aplikacje na wyższym poziomie. Być może ujemną cechą jest to, że względna łatwość użycia procesu powoduje, że programista nie zawsze jest świadomy kosztów z tym związanych. Jest to jednak właściwość większości konstrukcji wysokiego poziomu.

W proponowanym rozwiązaniu występuje dodatkowo konieczność (i możliwość) wskazania przez programistę komputera na którym ma być wykonywany nowo tworzony proces (patrz podręcznik użytkownika). Z jednej strony nakłada to na niego dodatkowy obowiązek, ale jednocześnie umożliwia mu kontrolę przydziału procesorów dla procesów. Niebagatelną zaletą takiego rozwiązania jest również znacznie łatwiejsza realizacja. Możliwe jest też dostarczenie programiście funkcji standardowej określającej numer komputera, na którym warto utworzyć kolejny proces. Oczywiście realizacja takiej operacji jest tak samo trudna jak realizacja systemu automatycznie przydzielającego procesor dla procesu, ale ma tę zaletę, że nie narzuca niczego programiście. Nie musi on korzystać z gotowych mechanizmów i może zrealizować własną strategię.

2. Mechanizmy komunikacji procesów rozproszonych

Brak wspólnej pamięci w systemach rozproszonych powoduje, że tradycyjne mechanizmy synchronizacji procesów, takie jak semafory czy warunkowe rejony krytyczne, nie mogą być użyte, gdyż opierają się na dzieleniu zmiennych przez procesy.

Podstawowym sposobem komunikacji w systemach rozproszonych jest przesyłanie komunikatów pomiędzy procesorami. Czasami takie komunikaty i odpowiednie operacje na nich są dostępne bezpośrednio w języku programowania wysokiego poziomu. Często jednak zamiast nich programista może posługiwać się innymi mechanizmami, na przykład zdalnym wołaniem procedur, przerwaniami lub spotkaniami. Jednak w każdym przypadku są one realizowane za pomocą komunikatów. Niekiedy pojedyncza akcja języka wysokiego poziomu odpowiada przesłaniu wielu komunikatów ([FF84], [G87]). W

opisywanej realizacji na przykład obce wołanie procedury wirtualnej wymaga przesłania czterech komunikatów.

Komunikacja procesów jest synchroniczna, gdy oba procesy rozpoczynają komunikację w tym samym momencie. Wtedy każdy z nich wie, w jakim stanie znajduje się ten drugi. O asynchronicznej komunikacji mówimy wtedy gdy procesy uczestniczą w niej niezależnie od siebie, jeśli chodzi o moment przystąpienia do komunikacji. W językach programowania wysokiego poziomu występują różne mechanizmy komunikacji, zazwyczaj jednak tylko jednego z obu wyżej wymienionych rodzajów. Czasami zdarzają się wyjątki, na przykład CHILL [R84] i Synchronizing Resources [FF84] posiadają zarówno mechanizmy synchroniczne jak i asynchroniczne. W zasadzie każdy program napisany z użyciem jednego z nich da się przepisać z użyciem drugiego i to z zachowaniem efektywności. Stwierdzenie to zostało udowodnione dla systemów z dzieloną pamięcią i istnieje uzasadnione przekonanie, że zachodzi również dla systemów rozproszonych [FF84]. W tym sensie mechanizmy komunikacji synchronicznej i asynchronicznej są równoważne, lecz nie znaczy to, że wygoda ich stosowania jest taka sama w przypadku każdego problemu. Właśnie konkretne zastosowanie powinno rozstrzygać o użyciu tego czy innego mechanizmu (języka programowania).

2.1. Komunikaty wysyłane asynchronicznie

Występują przede wszystkim w językach FLITS [F79] i CHILL [R84]. Komunikacja odbywa się za pomocą instrukcji operujących na komunikatach. Instrukcja:

send m **to** P

powoduje, że komunikat m zostaje przesłany do procesu o nazwie P. Z każdym procesem jest związana kolejka (najczęściej, choć nie zawsze, o nieograniczonej pojemności) zawierająca komunikaty wysłane do procesu ale jeszcze nie odebrane. Powyższa operacja wkłada więc komunikat m do kolejki związanej z procesem P. Proces P może odebrać komunikat za pomocą instrukcji:

receive m ,

która powoduje usunięcie pierwszego komunikatu z kolejki i przypisanie go zmiennej m.

W zależności od konkretnego języka programowania komunikaty mogą być tylko wartościami typów pierwotnych, sygnałami (tzn. komunikatami przekazującymi informację tylko poprzez sam fakt przesłania), rekordami lub obiektami dowolnego typu.

Mechanizm synchronizacji poprzez asynchroniczne komunikaty jest w swojej koncepcji najbardziej zbliżony do dostarczanego bezpośrednio przez sprzęt i oprogramowanie podstawowe (np. pakiety danych w systemie operacyjnym RSX-11 [R79]). Z tego względu jego realizacja jest stosunkowo łatwa. Wysiłek programistyczny stanowiący większą część mojej pracy byłby znacznie mniejszy, gdybym zdecydował się na taki mechanizm synchronizacji.

W porównaniu z proponowanym mechanizmem komunikaty asynchroniczne charakteryzują się małą mocą synchronizacji, co w pewnych przypadkach może prowadzić do występowania aktywnego czekania lub zmusza programistę do samodzielnego szeregowania żądań. W zamian za to zyskujemy większą swobodę i elastyczność oraz co za tym idzie, większą efektywność.

Za pomocą obcego wołania procedur i jawnej kolejki można osiągnąć efekt podobny do asynchronicznego przesłania komunikatu. Jest to możliwe dzięki temu, że proces wołany nie musi zawieszać się w oczekiwaniu na obce wołanie. Obce wołanie może zadziałać jak przerwanie zewnętrzne. Gdy ono nastąpi, odpowiednia wiadomość może zostać jedynie zakolejkowana. Gwarantuje to szybki powrót i prawie natychmiastowe wznowienie procesu wołającego. Proces wołany może tę wiadomość później odebrać (wyjąć z kolejki) w dowolnym momencie. Odpowiedni przykład jest zamieszczony w rozdziale IV.

Mechanizm komunikatów asynchronicznych można uogólnić wprowadzając pojęcie pojemnika (ang. container). Pojemnik jest abstrakcyjnym typem danych realizującym co najmniej operacje wstawiania i pobierania elementu. Operacje te oczywiście muszą mieć określone własności, ale nie precyzuje się ich dokładnie. Dzięki temu do pojemników można zaliczyć stosy, kolejki, listy, zbiory, wielozbiory. Zastępując w opisie mechanizmu komunikacji kolejki przez różnego rodzaju pojemniki można uzyskać różne, znane i nie znane, mechanizmy (np. semafory), nie zawsze zresztą sensowne. W przypadku symulacji przy użyciu obcego wołania procedur, odpowiednią strukturę danych wybiera sam programista, nie jest więc trudno zrealizować powyższą ideę bez zmiany języka programowania. Dzięki temu możliwe jest na przykład selektywne odbieranie komunikatów i sprawdzanie liczby komunikatów nie odebranych.

2.2. Przerwania asynchroniczne (sygnały)

Przerwania jako mechanizm komunikacji procesów były stosowane

najpierw w systemach czasu rzeczywistego. Klasycznym przykładem są przerwania AST (ang. Asynchronous System Traps) w systemie RSX-11 [R79]. Zadanie może za pomocą obsługi przerwań AST reagować na wiele różnych sytuacji, między innymi na otrzymanie komunikatu od innego zadania. W języku wysokiego poziomu przerwania pojawiły się chyba po raz pierwszy w PCS [PS84]. Występują one także w Loglanie 84 jako mechanizm dodatkowy (obok semaforów) [L87].

Komunikacja poprzez przerwania polega na tym, że jeden z procesów wykonując instrukcję:

interrupt P.H(e1, ..., en)

zmusza proces P do przerwania dotychczasowej działalności i rozpoczęcia wykonywania procedury H z przekazanymi parametrami (dozwolone są tylko parametry wejściowe). Proces przerywający wykonuje się dalej bez oczekiwania. Po zakończeniu procedury H proces P zostaje wznowiony w punkcie, w którym został przerwany. Dodatkowo proces przerywany może na pewien czas zablokować możliwość obsługi pewnych przerwań. Służą do tego odpowiedniki instrukcji **enable** i **disable** z proponowanego mechanizmu. Ponadto proces przerywający może za pomocą dodatkowego parametru instrukcji **interrupt** dowiedzieć się, czy przerwanie zostało obsłużone.

Jako samodzielny mechanizm komunikacji przerwania charakteryzują się jeszcze mniejszym stopniem synchronizacji niż poprzednio omawiane komunikaty. Wydaje się, że ten mechanizm jest uboższy niż obce wołanie procedur. Występuje tendencja do aktywnego oczekiwania zarówno w procesie przerywającym jak i przerywanym. Wynika to z tego, że nie istnieje żadna konstrukcja, która mogłaby spowodować zawieszenie procesu i zwolnienie procesora aż do momentu, gdy proces będzie gotowy do dalszego wykonywania. Uniknięcie aktywnego czekania jest niemożliwe bez dodatkowych mechanizmów. W Loglanie 84 przerwania wspomagane są przez tradycyjne semafory binarne. Jakkolwiek mechanizm ten jest zupełnie wystarczający, to nie nadaje się do realizacji w systemie rozproszonym.

Pewne rozszerzenia PCS przewidują instrukcje zapobiegające aktywnemu czekaniu [PS84] upodabniające mechanizm przerwań do obcego wołania procedur (odpowiadają one instrukcjom **call** i **accept** z proponowanego mechanizmu). Jednak zarówno przerwania w PCS jak i w obecnej wersji Loglanu 84 istotnie różnią się od obcego wołania procedur. W PCS i Loglanie rozpoczęcie obsługi przerwania nie powoduje zablokowania wszystkich sygnałów, natomiast w

proponowanym mechaniźmie wszystkie procedury zostają zablokowane. Właściwość blokowania przerwań w momencie rozpoczęcia obsługi przerwania jest cechą charakterystyczną wszystkich przerwań sprzętowych, a także wspomnianych wyżej przerwań AST. Brak tej cechy znacznie utrudnia synchronizację, gdyż stwarza możliwość niekontrolowanego zagnieżdżania się obsługi przerwań.

2.3. CSP

W zasadzie CSP [H78] jest raczej modelem komunikacji procesów niż samodzielnym językiem programowania. Mimo to miał duży wpływ na rozwój języków programowania równoległego (m.in. Ada [D83], Occam [EH87], Joyce [BH87]). Stanowi on swego rodzaju punkt odniesienia, z którym porównuje się różne języki i systemy, więc nie od rzeczy będzie wspomnieć o nim tutaj.

Podstawą mechanizmu komunikacji istniejącego w CSP jest idea wykonania instrukcji przypisania w taki sposób, że jeden proces dostarcza zmienną a drugi wyrażenie, którego wartość ma być obliczona i przypisana tej zmiennej. Wykonanie w procesie P instrukcji wyjścia:

Q!e

i w procesie Q instrukcji wejścia:

P?x

daje taki efekt jak instrukcja przypisania

x := e .

Synchronizację osiąga się poprzez fakt, że oba procesy uczestniczą w komunikacji jednocześnie. Proces, który jest pierwszy gotowy do komunikacji musi zaczekać aż ten drugi będzie również gotowy. Jest to więc komunikacja synchroniczna.

W porównaniu z CSP proponowany mechanizm jest bardziej elastyczny. Umożliwia on synchroniczną komunikację w obu kierunkach na raz, a nie tylko w jednym ustalonym kierunku. Poza tym proces wołany może dodatkowo wykonać pewną akcję w momencie komunikacji. Obie powyższe własności wynikają z tego, że podczas komunikacji proces wołany wykonuje pewną procedurę, na której zakończenie czeka proces wołający. Proces wołający może przekazać parametry wejściowe i odczytać parametry wyjściowe tej procedury.

Inną cechą odróżniającą proponowany mechanizm od CSP jest fakt, że proces wołany nie musi znać nazwy procesu wołającego. W specyficznym przypadku obcego wołania procedury formalnej nawet nazwa procesu wołanego pozostaje nieznana. Możliwe jest więc

napisanie procesu, który będzie obsługiwał żądania od wielu innych procesów, których liczba i nazwy nie są znane w trakcie pisania programu. Język Occam wywodzący się od CSP też ma taką możliwość, choć osiągniętą innymi środkami [EH87]. Identyfikacja procesu, z którym ma zajść komunikacja odbywa się tam nie bezpośrednio, ale przez podanie nazwy specjanego obiektu zwanego kanałem. Nazwa kanału może być znana obu procesom, nawet jeśli nie znają one nawzajem swoich nazw. Poza identyfikacją kanały nie mają żadnego wpływu na przebieg komunikacji.

W przypadku, gdy istotne jest rozróżnienie procesów wołających (tak jak w poniższym przykładzie) można każdemu z nich przydzielić osobną procedurę, za pomocą której będzie się komunikował.

W CSP występuje dodatkowo możliwość użycia instrukcji wejścia jako dozoru. W proponowanym mechaniźmie odpowiada jej instrukcja **accept** z kilkoma parametrami. Natomiast nie zawsze jest łatwe zasymulowanie występującej w CSP możliwości poprzedzenia takiego dozoru przez wyrażenia logiczne np.:

```
[ n > 0;  P?x -> ...
| n >= 0; Q?x -> ...
| n < 0;  R?x -> ...
]
```

Jeśli co najmniej dwa z warunków logicznych mogą być spełnione jednocześnie (i nie chcemy przypisać priorytetu żadnemu z nich), to przepisując powyższy fragment programu z użyciem obcego wołania procedur należy wyliczyć explicite wszystkie możliwe podzbiory spełnionych warunków:

```
if n > 0
then accept P, Q
else
  if n = 0
  then accept Q
  else accept R
  fi
fi
```

Trzeba jednak zaznaczyć, że jest to metoda bardzo ogólna. W wielu konkretnych przypadkach można łatwo uniknąć takiego wyliczania rozwiązując problem od początku z użyciem obcego wołania procedur. Szczególnie przydatne są wtedy operacje na masce odblokowanych procedur.

Najważniejszą różnicą w stosunku do CSP jest jednak możliwość obcego wywołania procedury w przypadku, gdy proces wołany wykonuje

jakieś akcje niekoniecznie związane z komunikacją. Proces może być gotowy do reakcji na jakieś zdarzenie, ale nie musi na nie oczekiwać. Z punktu widzenia procesu wołanego obce wywołanie *procedury* działa jak przerwanie. Istnieje więc możliwość stosunkowo prostej realizacji asynchronicznej obsługi żądań. W wielu przypadkach zmniejsza to liczbę procesów potrzebnych do rozwiązania konkretnego problemu.

2.4. Spotkanie w BNR Pascalu [687]

Jest to mechanizm najbardziej zbliżony do proponowanego, nawet jeśli chodzi o składnię (chociaż powstał niezależnie). Do synchronizacji procesów służą instrukcje wywołania procedury:

call X.P(...)

gdzie X jest nazwą procesu oraz instrukcja oczekiwania na wywołanie procedury:

accept [P1, ..., Pn].

Semantyka tych instrukcji jest dokładnie taka jak w przypadku obcego wołania procedur, jeśli się usunie instrukcje **enable**, **disable**, **return enable** ... **disable**. Dzięki brakowi instrukcji operujących na masce procedur upraszcza się opis i realizacja. W BNR Pascalu samo pojęcie maski odblokowanych procedur jest niepotrzebne, gdyż proces wołany może przyjąć obce wołanie procedury tylko podczas wykonywania instrukcji **accept**. W innych momentach wszystkie procedury są zablokowane. Jest to więc ściśle synchroniczne spotkanie w swojej najczystszej postaci.

Zrealizowany mechanizm obcego wołania procedur jest pewnym rozszerzeniem spotkania z BNR Pascala. Ma on tę przewagę, że pozwala na bardzo łatwą symulację asynchronicznej obsługi żądań. Można to osiągnąć korzystając z obcego wołania tak jak z przerwań. Wtedy symulacja nie wymaga wprowadzania dodatkowego zadania buforującego, co byłoby konieczne w BNR Pascalu. Wydaje się, że mechanizm czystego spotkania, w istocie bardzo elegancki, jest nieco za słaby w praktyce (choć BNR Pascal był stworzony z myślą o konkretnym zastosowaniu, więc może nie mam racji). Brakuje w nim przede wszystkim możliwości warunkowej obsługi żądań (por. przykład semafora w rozdziale IV).

Ceną za dodatkowe możliwości, jaką trzeba zapłacić w przypadku obcego wołania procedur, jest utrata elegancji i prostoty. Proponowany mechanizm ma charakter raczej niskopoziomowy (przynajmniej w porównaniu ze spotkaniem). Wynika to z tego, że

instrukcje **enable/disable** nie są strukturalne. Nie można na podstawie samego tekstu programu stwierdzić, w których fragmentach jest możliwe przyjęcie obcego wywołania danej procedury. Także instrukcje **return enable ... disable** wywołują wrażenie bezpośredniego odwzorowania mechanizmów sprzętowych. Są to jednak operacje bardzo użyteczne i ich obecność jest chyba uzasadniona. Wydaje się, że stanowią one rozsądny kompromis pomiędzy elegancją a elastycznością.

2.5. Spotkanie w Adzie [DS83]

Choć ma ono inną składnię, jest też rozszerzeniem czystego spotkania (takiego jak w BNR Pascalu), ale idącym w innym kierunku. Jednym z głównych zastosowań Ady miało być programowanie systemów wbudowanych (ang. embedded systems), bardzo często pracujących w czasie rzeczywistym. Wprowadzono więc bardzo silną instrukcję **select**, która pozwala na bardzo wyszukaną warunkową obsługę żądań.

W najprostszym przypadku służy ona do oczekiwania na komunikację poprzez jedno z kilku wejść:

```
select
  accept P1(...) do ... end
or
  ...
or
  accept Pn(...) do ... end
end select;
```

Powyższa instrukcja zawiesza zadanie do momentu, gdy jakieś inne zadanie wywoła jedno z wejść Pi. W proponowanym mechanizmie odpowiada jej wtedy instrukcja **accept** z kilkoma parametrami, ale na tym podobieństwo się kończy.

Podobnie jak w CSP instrukcje **accept** mogą być poprzedzone wyrażeniem logicznym określającym warunki, które muszą być spełnione aby komunikacja mogła zajść:

```
select
  when B1 => accept P1(...) do ... end
or
  ...
or
  when Bn => accept Pn(...) do ... end
end select;
```

Inne formy instrukcji **select** pozwalają na zaniechanie komunikacji, gdy nie jest ona możliwa natychmiast, i to zarówno po stronie wołającej jak i wołanej:

```
select
  accept P(...) do ... end
else
  ...
end select;
```

(podobnie dla instrukcji wołania wejścia). W przypadku, gdy komunikacja nie jest możliwa natychmiast, wykonują się instrukcje po **else**.

Instrukcja **select** w połączeniu z instrukcją **delay** pozwala na obsługę lub wywoływanie wejść z możliwością przeterminowania po ustalonym czasie. Fragment programu:

```
select
  accept P(...) do ... end      -- ew. wołanie wejścia
else
  delay 15.0
end select;
```

powoduje, że komunikacja zostanie zaniechana, jeżeli nie będzie ona możliwa w ciągu 15 sekund od chwili rozpoczęcia wykonywania instrukcji **select**.

Oprócz tego programista może traktować wystąpienie przerwania sprzętowego jako wywołanie odpowiedniego wejścia w swoim zadaniu. Istnieje także możliwość zapytania o liczbę zadań czekających na obsługę danego wejścia, co często ułatwia uniknięcie zagłodzenia (choć ze względu na przeterminowania, liczba czekających zadań może czasami ulec zmniejszeniu).

Wszystkie powyższe udogodnienia powodują znaczne problemy realizacyjne i konieczność używania skomplikowanych protokołów komunikacji między zadaniami ([FF84], [G87]). W zamian za to zyskuje się elastyczność i szeroki zakres zastosowań. Spotkanie w Adzie na pewno góruje nad obcym wołaniem procedur, jeśli chodzi o zastosowania w systemach czasu rzeczywistego. Pojawiające się tam problemy bardzo trudno jest rozwiązywać nie dysponując możliwością przeterminowania i pojęciem czasu. Rozszerzenie proponowanego mechanizmu w odpowiednim kierunku jest możliwe, przynajmniej w ograniczonym zakresie. W pełnej ogólności byłoby trudne ze względu na możliwość odwołania obcego wołania procedury. Przy nieuważnej realizacji odwołanie mogłoby nastąpić po rozpoczęciu wykonywania tej procedury w procesie wołanym. Natomiast przeterminowanie

instrukcji **accept** można by zrealizować stosunkowo prosto, gdyż proces wykonujący instrukcję **accept** nie wysyła żadnych komunikatów, więc zaniechanie oczekiwania nie ma konsekwencji dla innych procesów. Nawet w tym przypadku brak wsparcia ze strony systemu operacyjnego jest jednak wyraźnym utrudnieniem.

Obce wołanie procedur ma jednak pewną przewagę nad mechanizmami występującymi w Adzie. Podobnie jak w BNR Pascalu spotkanie w Adzie nie pozwala na definiowanie wejść, które mogą być wywołane asynchronicznie, tzn. bez oczekiwania ze strony wołanego zadania. Niedogodność ta jest co prawda mniejsza niż w przypadku BNR Pascala ze względu na opisane wyżej rozszerzenia. Nadal jednak można podać przykłady problemów, których rozwiązanie przy użyciu obcego wołania procedur wymaga mniejszej liczby procesów niż analogiczne rozwiązanie zapisane w Adzie.

2.6. Zdalne wołanie procedur w Distributed Processes [BH78]

Mechanizm ten zaproponowany przez Brinch Hansena, podobnie jak obce wołanie procedur opiera się na wywoływaniu procedur (wejść) wykonywanych przez proces wołany. Istotna różnica polega na tym, że w Distributed Processes synchronizację i szeregowanie uzyskuje się za pomocą instrukcji dozorowanych zawieszających właśnie wykonywaną procedurę. W istocie Distributed Processes jest rozproszoną wersją warunkowych rejonów krytycznych, używanych czasami w systemach z dzieloną pamięcią. Oto przykład procesu realizującego semafor, zapisanego w Distributed Processes:

```
process semaphore;
var n : integer;
  procedure P;
  begin
    when
      n > 0 : n := n-1
    end
  end;
  procedure V;
  begin
    n := n+1;
  end;
begin
  n := 0;
end
```

Po uruchomieniu programu w Distributed Processes każdy proces zaczyna wykonywać swoje instrukcje do momentu, gdy zostanie zawieszony w instrukcji **when** z niespełnionym dozorem. Może wtedy zostać przerwany i zacząć wykonanie procedury wywołanej przez inny proces. Jej wykonanie także może zostać przerwane przez zawieszenie w instrukcji dozorowanej. Za każdy razem, gdy proces zostaje zawieszony z powodu niespełnionego dozoru, wybiera się i wznawia jedną z uprzednio zawieszonych instrukcji mających w danym momencie spełniony dozór. Zamiast tego można też rozpocząć wykonanie nowej procedury wywołanej przez inny proces. Tak więc wykonywanie własnej treści procesu i zdalnie wywoływanych procedur jest przeplatane, ale nie ma wywłaszczania. Przekazanie sterowania może nastąpić jedynie na skutek zawieszenia w instrukcji dozorowanej.

W przypadku obcego wołania procedur synchronizacja następuje przed wywołaniem poprzez blokowanie lub odblokowywanie możliwości wykonania procedury, przy czym są używane zwykłe instrukcje warunkowe. Wydaje się, że zdalne wołanie procedur w Distributed Processes jest bardziej wygodne i elastyczne, przede wszystkim dlatego, że w dozorach mogą wystąpić parametry właśnie wykonywanej procedury. Często zmniejsza to liczbę wywołań potrzebnych do osiągnięcia właściwej synchronizacji. Ma to miejsce wtedy, gdy proces usługowy obsługuje żądania nie w kolejności ich nadchodzenia, ale zależnie od priorytetu lub innych parametrów. Sytuacja taka występuje w przykładzie 3 w rozdziale IV.

Możliwość użycia dowolnych wyrażeń logicznych jako dozorów, chociaż wygodna, powoduje niestety, że realizacja Distributed Processes jest nieefektywna, gdyż wymaga ciągłego sprawdzania być może skomplikowanych warunków. Można wprawdzie tego uniknąć przeprowadzając analizę wyrażenia i reagując na zmiany wartości tylko tych zmiennych, od których zależy jego wartość. W praktyce jest to jednak rozwiązanie bardzo skomplikowane ([S76], [IM82]). Próby wspomożenia kompilatora poprzez zmiany samego mechanizmu prowadzą do koncepcji tzw. zdarzeniowych rejonów krytycznych [IM82]. Nie będą one tu omówione, gdyż ich znacznie bardziej elegancką wersję stanowią monitory.

2.7. Monitory

Występują one w wielu językach programowania współbieżnego, w których istnieją zmienne dzielone, m.in. w Moduli [W77],

Concurrent Pascalu [BH75], CHILL [R84]. Monitor łączy w sobie funkcje synchronizacji dostępu do zasobów i modułu ukrywającego w swoim wnętrzu szczegóły realizacji operacji na tych zasobach:

```
monitor M;
var x : T;            (* deklaracja zmiennej dzielonej *)
    c : condition;    (* deklaracja zmiennej warunkowej *)
  entry procedure P1(...);
    ...
    delay(c);
    ...
  end P1;
  entry procecure Pn(...);
    ...
    continue(c);
    ...
  end Pn;
begin
  (* nadanie wartości początkowych *)
end M;
```

Wszystkie zmienne dzielone używane przez procesy muszą być umieszczone w jakimś monitorze. Dostęp do tych zmiennych jest dozwolony tylko poprzez wywołania specjalnych procedur monitora. W danym momencie tylko jeden proces może wykonywać procedurę danego monitora. Gdy inny proces chce wykonać procedurę tego samego monitora, zostanie zawieszony do momentu, w którym pierwszy proces zwolni monitor. Wzajemne wykluczanie przy dostępie do zmiennych dzielonych jest więc zagwarantowane automatycznie.

Ponadto wewnątrz monitora jest dostępny typ standardowy **condition**. Z każdą zmienną typu **condition** związana jest kolejka procesów. Na zmiennych tego typu dozwolone są tylko trzy operacje. Operacja **delay**(c) powoduje, że wykonujący ją proces zawiesza się w kolejce związanej ze zmienną warunkową c. Jednocześnie zwalnia on dostęp do monitora pozwalając innemu procesowi rozpocząć wykonywanie procedury wejściowej. Operacja **continue**(c) powoduje, że pierwszy proces z kolejki c (jeśli kolejka nie jest pusta) zostaje wznowiony. Proces wykonujący operację zostaje zawieszony w oddzielnej kolejce o najwyższym priorytecie. Dodatkowy predykat **empty**(c) sprawdza pustość kolejki związanej ze zmienną c.

Monitory są mechanizmem bardzo wygodnym i eleganckim. W

porównaniu do proponowanego mechanizmu mają zalety podobne jak opisane powyżej zdalnie wołane procedury w Distributed Processes. Zawieszanie procesów wykonujących procedury monitora może zależeć warunkowo od parametrów wołanej procedury. Ponadto monitory posiadają jeszcze jedną ważną zaletę. Mianowicie można je stosunkowo łatwo zrealizować efektywnie (całkowicie bez aktywnego czekania).

Niestety zgodnie ze swoją ścisłą definicją monitory nie nadają się do synchronizacji procesów rozproszonych. Wynika to z tego, że procedury monitora mają dostęp do wspólnych struktur danych, a są wykonywane przez procesy wołające. Można jednak zauważyć, że w każdym momencie wykonywana jest co najwyżej jedna procedura monitora. Modyfikując nieco semantykę, a raczej sposób opisu, można zrealizować monitory jako procesy. Proces będący monitorem wykonuje procedury wywoływane przez inne procesy, sam sięgając tylko do swoich zmiennych lokalnych.

Wydawałoby się, że w ten sposób można stosunkowo łatwo wydefiniować monitory przy użyciu obcego wołania procedur, tym bardziej, że w proponowanym mechaniźmie wzajemne wykluczanie wołanych z zewnątrz procedur jest zagwarantowane automatycznie. Okazuje się jednak, że nie jest to takie proste, jeśli chce się zrealizować operacje **delay** i **continue**. Operacje te mogą spowodować, że wykonywanie procedury monitora zostanie przerwane na jakiś czas, w którym będzie wykonywana inna procedura, być może też tylko częściowo. Tak więc z punktu widzenia procesu-monitora procedury wejściowe zachowują się raczej jak współprogramy, a nie jak procedury przez niego wykonywane. Z tego względu bezpośrednie symulowanie monitorów za pomocą obcego wołania procedur nie jest ani łatwe ani eleganckie. Nie zostanie ono przedstawione, gdyż nie wydaje się być najlepszym sposobem korzystania z proponowanego mechanizmu.

ROZDZIAŁ IV

Przykłady użycia obcego wołania procedur w typowych problemach synchronizacji procesów.

1. Realizacja semafora jako procesu

1.1. Rozwiązanie przy użyciu obcego wołania jako czystego spotkania

```
unit semafor:process(node,n:integer);
(* node oznacza numer węzła w sieci (patrz podręcznik
użytkownika), n - wartość początkową semafora *)

  unit P:procedure;
  (* procedura obsługuje operację P na semaforze - zmniejsza *)
  (* wartość licznika semafora. Wywołanie możliwe tylko dla n > 0 *)
  begin
    n := n-1;
  end P;

  unit V:procedure;
  (* operacja V - zwiększenie licznika. Wywołanie zawsze możliwe. *)
  begin
    n := n+1;
  end V;

begin
  return;                  (* powrót z tworzenia obiektu procesu *)
  (* oczekujemy w pętli na wykonanie operacji na semaforze *)
  do
    if n > 0               (* jeśli wartość semafora jest dodatnia *)
    then                   (* można wykonać operację P lub V *)
      accept P, V
    else                   (* w przeciwnym przypadku tylko V *)
      accept V
    fi;
  od
end semafor;
...
var s:semafor;             (* deklaracja semafora *)
```

```
...
begin
  (* utworzenie semafora i nadanie wartości początkowej *)
  s := new semafor(node, 1);
  resume(s);
  ...
  (* wykonanie operacji P na semaforze *)
  call s.P;
  ...
  (* sekcja krytyczna *)
  ...
  (* wykonanie operacji V *)
  call s.V;
  ...
end
```

Przykład ten dowodzi, że obce wołanie procedur jest co najmniej tak silnym mechanizmem synchronizacji jak semafory, a więc także inne klasyczne mechanizmy (np. warunkowe rejony krytyczne).

W tym przypadku nie są używane instrukcje zmieniające maskę odblokowanych procedur inne niż **accept**. Z mechanizmu obcego wołania procedury korzysta się wtedy tak jak z czystego spotkania. Proces, który pierwszy będzie chciał się skomunikować musi zaczekać do momentu, gdy drugi proces też będzie gotowy. Komunikacja (poprzez wykonanie procedury) odbędzie się dopiero wtedy, gdy oba procesy będą tego chciały. Powyższe rozwiązanie w dokładnie taki sam sposób daje się zapisać w BNR Pascalu czy w Adzie.

1.2. Rozwiązanie z użyciem instrukcji zmieniających maskę odblokowanych procedur

```
unit semafor:process(node,n:integer);
(* node oznacza numer węzła w sieci, n - wartość początkową *)

  unit P:procedure;
  (* Operacja P na semaforze. *)
  begin
    n := n-1;              (* zmniejszamy licznik semafora *)
    if n=0                 (* jeśli zmniejszyliśmy do 0 *)
```

```
    then                    (* to blokujemy dalsze obce wywołania *)
      return disable P      (* procedury P *)
    fi
  end P;

  unit V:procedure;
  (* Operacja V *)
  begin
    n := n+1;               (* zwiększamy licznik *)
    return enable P         (* licznik na pewno większy niż 0, więc *)
                            (* odblokowujemy wywołania procedury P *)
  end V;

begin
  return;
  (* odblokowujemy procedurę V, gdyż operacja V jest *)
  (* zawsze dozwolona *)
  enable V;
  (* procedurę P odblokowujemy jeśli wartość początkowa *)
  (* semafora jest większa niż 0. Jeśli nie to odblokowanie *)
  (* procedury P może nastąpić jedynie poprzez wywołanie *)
  (* procedury V *)
  if n > 0
  then enable P fi;
  (* oczekujemy w pętli na wywołanie którejkolwiek z aktualnie *)
  (* odblokowanych procedur *)
  do
    accept ?
  od
end semafor;
```

To rozwiązanie różni się od poprzedniego tym, że procedury P i V obsługujące operacje semaforowe synchronizują się same poprzez operacje wykonywane na masce odblokowanych procedur. Dzięki temu nie jest potrzebna synchronizacja poprzez instrukcje zawarte w treści procesu semafora. Taki proces może właściwie zajmować się czymś zupełnie innym. Nie da się tego uzyskać w przypadku ściśle synchronicznego spotkania (takiego jak w poprzednim rozwiązaniu). Możliwe jest nawet zrealizowanie dowolnej ustalonej liczby semaforów za pomocą jednego procesu. W tym celu należy tylko odpowiednią liczbę razy powtórzyć deklaracje procedur P i V (np.

korzystając z prefiksowania) oraz parametru n (oczywiście z nowymi nazwami). Jest to także możliwe w Adzie ponieważ w instrukcji **select** mogą występować wyrażenia logiczne jako dozory. Realizacja dwóch semaforów jako zadania w Adzie wyglądałaby mniej więcej tak:

```
loop
  select
    when n1 > 0 => accept P1 do n1 := n1-1 end
  or
    accept V1 do n1 := n1+1 end
  or
    when n2 > 0 => accept P2 do n2 := n2-1 end
  or
    accept V2 do n2 := n2+1 end
  end select
end loop
```

W BNR Pascalu musielibyśmy wypisać explicite warunki uwzględniające wszystkie możliwe kombinacje podniesionych i opuszczonych semaforów.

2. Problem producent-konsument

Rozwiązanie zostanie przedstawione w przypadku wielu producentów i jednego konsumenta. Aby uczynić problem bardziej konkretnym załóżmy, że mamy do czynienia z fragmentem systemu operacyjnego. Proces obsługujący drukarkę przyjmuje od innych procesów żądania wydrukowania wskazanych plików. Proces wysyłający żądanie nie czeka na jego realizację. Nie interesuje go również fakt, że plik został już wydrukowany. Ważne jest natomiast aby nie musiał czekać, jeżeli tylko kolejka procesu obsługującego drukarkę nie jest całkowicie zapełniona.

```
unit queue:class(type element; size:integer);
(* Pomocnicza klasa realizująca kolejki o ograniczonej pojemności.
   Element oznacza typ elementu kolejki, size - maksymalny rozmiar
   kolejki *)

  unit insert:procedure(e:element); ...
    (* wstawienie elementu do kolejki *)
  unit delete:function:element; ...
```

```
    (* pobranie i usunięcie elementu z kolejki *)
  unit empty:function:boolean; ...
    (* sprawdzenie, czy kolejka jest pusta *)
  unit full:function:boolean; ...
    (* sprawdzenie, czy kolejka jest całkowicie zapełniona *)

end queue;
...
unit spooler:process(node:integer);
(* Proces obsługujący drukarkę. Żądanie wydrukowania pliku *)
(* przesyła się wywołując procedurę print *)

var
  Q:queue,                  (* kolejka plików do wydrukowania *)
  f:filename,

 unit print:procedure(f:filename);
 (* Procedura przyjmuje żądanie wydrukowania pliku o nazwie f. *)
 (* Jej wywołanie jest możliwe tylko wtedy, gdy kolejka Q nie *)
 (* jest całkowicie zapełniona *)
 begin
   call Q.insert(f);        (* kolejkujemy żądanie *)
   if Q.full                (* jeśli zapełniliśmy kolejkę to dalsze *)
   then                     (* obce wywołania print blokujemy *)
     return disable print
   fi;
 end print;

begin
  (* utworzenie kolejki plików do wydrukowania *)
  Q := new queue(filename, 50);
  return;
  do
    (* pobieramy z kolejki pierwszy plik *)
    disable print;          (* wejście do sekcji krytycznej *)
    if Q.empty              (* jeśli kolejka jest pusta *)
    then
      accept print          (* to czekamy na jakieś żądanie *)
    fi;
    f := Q.delete;          (* pobieramy pierwszy element *)
    enable print;           (* kolejka nie jest pełna, można przyjąć *)
```

```
                    (* żądanie *)
    (* teraz drukujemy plik o nazwie f *)
    ...
  od
end spooler;
```

Przykład ten pokazuje w jaki sposób za pomocą obcego wołania procedur można uzyskać efekt asynchronicznego wysłania komunikatu. Dzięki temu, że proces wołany nie musi oczekiwać na obce wołanie, przyjęcie wołania nie wymaga aby proces wołany znajdował się w ustalonym stanie. Kolejkę komunikatów (w tym wypadku nazw plików) można obsługiwać bezpośrednio w procesie realizującym żądania (drukującym). W przypadku spotkania konieczne byłoby zastosowanie dodatkowego zadania buforującego nadchodzące komunikaty.

Warto dodać, że w przypadku kolejki o nieograniczonej pojemności można usunąć instrukcję **if** Q.full **then** ... z procedury print. Wtedy symulowana jest dokładnie komunikacja poprzez asynchronicznie wysyłane komunikaty (takie jak np. w PLITS [F79]). Instrukcja **call** spool.print(f) odpowiada instrukcji

send f **to** spool ,

natomiast ciąg instrukcji:

```
    disable print;          (* wejście do sekcji krytycznej *)
    if Q.empty              (* jeśli kolejka jest pusta *)
    then
      accept print          (* to czekamy na jakieś żądanie *)
    fi;
    f := Q.delete;          (* pobieramy pierwszy element *)
    enable print;           (* wyjście z sekcji krytycznej *)
```

odpowiada instrukcji **receive** f. Używając różnych nazw dla procedur odpowiadających różnym rodzajom komunikatów można wydefiniować występujący w PLITS mechanizm selektywnego (ze względu na rodzaj) wybierania komunikatów z kolejki wejściowej. Natomiast odbieranie komunikatów połączone z selekcją nadawcy wymaga, aby każdy nadawca używał innej procedury do przesyłania swoich komunikatów. Nie jest to sposób tak bezpieczny jak w PLITS, gdyż stwarza możliwość przesłania komunikatu z cudzym podpisem (poprzez wywołanie nieodpowiedniej procedury).

Zupełnie analogicznie można rozwiązać problem dualny: jeden producent i wielu konsumentów. Producent wkłada kolejne produkty

do kolejki w nim zadeklarowanej. Procesy konsumentów wywołują procedurę w procesie producenta, która wyjmuje produkt z kolejki i przekazuje poprzez parametr wyjściowy. W tym wypadku nawet dysponując możliwością asynchronicznego wysyłania komunikatów nie da się uniknąć wprowadzenia dodatkowego procesu buforującego. Dopiero możliwość przerwania innego procesu i wymuszenia na nim wykonania jakiejś akcji pozwala na buforowanie produktów bezpośrednio przez proces producenta.

Natomiast w przypadku wielu producentów i wielu konsumentów rzeczywiście trzeba przewidzieć proces-bufor, który będzie przechowywał wyprodukowane a nie skonsumowane produkty. Najprostsze rozwiązanie nie różni się od tego jakie byłoby potrzebne w Adzie.

3. Problem czytelników i pisarzy

Rozwiązanie przy użyciu obcego wołania procedur jest w najprostszym przypadku zupełnie łatwe (podobnie zresztą jak w Adzie). Przedstawione rozwiązanie różni się tym, że zapobiega zagłodzeniu zarówno czytelników jak i pisarzy. Jest to osiągnięte poprzez naprzemienną obsługę pisarza i grup czytelników. Oczywiście w przypadku braku konfliktu czytanie lub pisanie jest możliwe natychmiast.

```
unit controller:process(node:integer);
(* Proces zarządzający, do którego kierowane są żądania
dopuszczenia do czytania lub pisania *)
var
  waiting_readers:integer,   (* liczba oczekujących czytelników *)
  writing:boolean,           (* TRUE jeśli pisarz w sekcji kryt. *)
  (* poniższe zmienne dotyczą pojedyńczej grupy czytelników *)
  cr:integer,                (* całkowita liczba czytelników *)
  br:integer,                (* liczba czytelników, którzy *)
                             (* rozpoczęli czytanie *)
  er:integer;                (* liczba czytelników, którzy *)
                             (* zakończyli czytanie *)

  unit stamp:procedure;
  (* Procedura wywoływana przez czytelnika bezpośrednio przed *)
  (* żądaniem umożliwienia czytania. Pozwala ona zliczać *)
  (* oczekujących czytelników *)
```

```
  begin
    waiting_readers := waiting_readers+1;
  end stamp;

  unit startread:procedure;
  (* Obsługuje rozpoczęcie czytania przez czytelnika *)
  begin
    br := br+1;          (* kolejny czytelnik rozpoczął czytanie *)
    writing := false;    (* pisarz nie jest w sekcji krytycznej *)
  end startread;

  unit endread:procedure;
  (* Zakończenie czytania *)
  begin
    er := er+1;          (* jeden czytelnik zakończył czytanie *)
  end endread;

  unit startwrite:procedure;
  (* Rozpoczęcie pisania przez pisarza *)
  begin
    writing := true;     (* pisarz w sekcji krytycznej *)
  end startwrite;

  unit endwrite:procedure;
  (* Zakończenie pisania. *)
  begin
    (* nie wykonuje akcji, ma znaczenie tylko synchronizacyjne *)
  end endwrite;

begin
  return;
  (* umożliwiamy rejestrację oczekujących czytelników *)
  enable stamp;
  do
    (* rozpoczynamy kolejny cykl obsługi *)
    br := 0;  er := 0;

    (* wpuszczamy do sekcji krytycznej pierwszego czytelnika *)
    (* lub pisarza (mechanizm obcego wołania gwarantuje nam *)
    (* sprawiedliwość) *)
    accept startread, startwrite;
```

```
    if writing              (* jeśli był to pisarz *)
    then                    (* to czekamy, aż skończy pisać *)
      accept endwrite;      (* i kończymy cykl obsługi *)
    else
      (* jeśli był to czytelnik, to badamy ilu ich jeszcze *)
      (* czeka na dopuszczenie do czytania *)
      disable stamp;
      cr := waiting_readers+1;   (* liczba oczekujących czyt. *)
      waiting_readers := 0;      (* kolejni będą włączeni do *)
                                 (* następnej grupy czytelników *)
      enable stamp;

      (* wpuszczamy do sekcji krytycznej wszystkich aktualnie *)
      (* oczekujących czytelników, umożliwiając jednocześnie *)
      (* kończenie czytania *)
      while br < cr
      do
        accept startread, endread;
      od;
      (* czekamy, aż reszta czytelników skończy czytać *)
      while er < cr
      do
        accept endread
      od;
      (* koniec cyklu obsługi *)
    fi;
  od
end controller;
...
unit reader:process(node:integer, ctrl:controller);
(* Szkielet procesu czytelnika. Ctrl oznacza wskaźnik do procesu
zarządzającego *)
begin
  return;
  do
    call ctrl.stamp;      (* rejestracja oczekującego czytelnika *)
    call ctrl.startread;  (* żądanie rozpoczęcia czytania *)
    ...
    (* czytanie *)
    ...
    call ctrl.endread;    (* zakończenie czytania *)
```

```
    ...
  od
end reader;

unit writer:process(node:integer, ctrl:controller);
(* Proces pisarza *)
begin
  return;
  do
    call ctrl.startwrite;  (* żądanie rozpoczęcia pisania *)
    ...
    (* pisanie *)
    ...
    call ctrl.endwrite;    (* zakończenie pisania *)
   ...
  od
end writer;
...
begin (* program główny *)
 (* najpierw uruchamiamy proces zarządzający *)
 c := new controller(0);
 resume(c);
 (* następnie uruchamiamy pewną liczbę procesów czytelników i *)
 (* pisarzy przekazując im jako parametr wskaźnik do procesu *)
 (* zarządzającego *)
 for i := 1 to num_readers
 do
   resume(new reader(1, c));
 od;
 for i := 1 to num_writers
 do
   resume(new writer(2, c));
 od;
end
```

Widać wyraźnie, że rozwiązanie byłoby prostsze, gdyby zrealizowany mechanizm synchronizacji bezpośrednio udostępniał liczbę procesów oczekujących na przyjęcie obcego wołania danej procedury. Niepotrzebna byłaby wtedy procedura stamp w procesie zarządzającym. Rozszerzenie proponowanego mechanizmu w tym kierunku jest możliwe poprzez dodanie odpowiedniej operacji standardowej.

ROZDZIAŁ V

Opis realizacji obcego wołania procedur.

1. Wstęp

Realizacja została wykonana poprzez wprowadzenie zmian do istniejącego systemu Loglanu 82 na komputerach IBM PC. Jest to bardzo duży produkt programistyczny, liczący obecnie ponad 40000 linii kodu źródłowego, z tego większość w Fortranie. Tylko do niektórych fragmentów systemu istnieje dokumentacja. Spowodowało to liczne trudności wynikające po prostu z braku wiedzy, której nie było skąd uzyskać. W tym miejscu należą się podziękowania członkom zespołu LOGLAN, a w szczególności pani Danucie Szczepańskiej, za nieocenioną pomoc w rozwikływaniu trudnych zagadek detektywistycznych. Jest jednak bardzo prawdopodobne, że istnieją jeszcze nie wykryte błędy spowodowane właśnie takim stanem rzeczy (poza oczywiście moimi błędami projektowymi).

2. Cele eksploatacyjne.

Realizacja została wykonana przy użyciu sieci lokalnej komputerów IBM PC, która została zainstalowana w Instytucie Informatyki na wiosnę 1987 roku. Komputery są połączone w sieć za pomocą kart D-Link firmy Datex Technology Inc.. Dzięki użyciu sieci możliwe było zrealizowanie prawdziwej, nie symulowanej, równoległości.

Wymaganie, aby każdy proces był wykonywany przez oddzielny procesor w postaci komputera IBM PC, wydawało się zbyt mocne z dwóch powodów. Po pierwsze, drastycznie ograniczałoby to liczbę tych użytkowników Loglanu, którzy uruchamialiby programy równoległe. Jakkolwiek nie jest jasne, czy w ogóle tacy będą, to nie należy z góry wykluczać tych, którzy nie posiadają akurat sieci typu D-Link. Po drugie, liczba procesów w programie równoległym musiałaby być nie większa niż liczba komputerów połączonych w sieć (w naszych warunkach jest to zazwyczaj liczba jednocyfrowa). Poza tym uruchamianie programu równoległego na dużej liczbie komputerów jest niewygodne, szczególnie gdy są one oddalone od siebie.

Z powyższych powodów podjąłem decyzję o zrealizowaniu również równoległości symulowanej na jednym komputerze. Spowodowało to

rozpoznawanie. Nowe słowa kluczowe są związane następująco:

enable klasa 35 numer 0

disable klasa 35 numer 1

accept klasa 36 numer 0

Oprócz tego w tablicy HASH znajduje się nie używany identyfikator QUIT, który pierwotnie miał być jeszcze jednym słowem kluczowym. Pomysł ten został zaniechany po wprowadzeniu opcji **enable/disable** przy instrukcji **return**. Niestety już nie można usunąć tego identyfikatora z tablicy!

5.2. Analiza składniowa

W tej części kompilatora wszystkie zmiany dotyczyły procedury E8 (WANZ.FOR) rozpoznającej instrukcje. Musiały one uwzględnić nowe instrukcje **enable**, **disable**, **accept** oraz opcje **enable/disable** przy instrukcji **return**.

Dodano zmienne (lokalne dla E8), których wartościami są kody nowych symboli jednostek leksykalnych i kodu pośredniego:

WENAB, WDISAB, WACCEP, WPREND, SENAB.

Obsługę instrukcji **return** zmieniono dodając rozpoznawanie opcji **enable/disable**. Kod pośredni (przekazywany do przebiegu analizy semantycznej) generowany dla instrukcji **return** wygląda teraz następująco:

WRETURN ((WENAB | WDISAB) (WIDENT identyfikator)+)* WPREND

Dodano rozpoznawanie instrukcji **enable/disable** (klasa 35). Generowany jest kod:

(WENAB | WDISAB) (WIDENT identyfikator)+ WPREND

Dla instrukcji **accept** (klasa 36) generowany jest kod:

WACCEP (WIDENT identyfikator)* WPREND

5.3. Semantyka statyczna

W procedurze PROTP1 (DSW.FOR) dodano częściową kontrolę nagłówka procesu. Sprawdzane jest czy:

- pierwszy parametr pierwszego procesu w łańcuchu prefiksowym (oznaczający numer węzła) jest typu INTEGER,
- pozostałe parametry są jednego z typów pierwotnych lub procesowych, procedurami lub funkcjami formalnymi,
- suma apetytów parametrów nie przekracza pojemności jednego komunikatu. Wynikające stąd ograniczenie na liczbę parametrów zależy od modelu pamięci, typów parametrów oraz ich rodzaju (tzn.

sporo trudności, ale było warte włożonego wysiłku.

Drugim istotnym celem było udostępnienie uzyskanych rezultatów użytkownikom najnowszej wersji kompilatora Loglanu na IBM PC, a nie jakiejś nie pielęgnowanej wersji archiwalnej. Stało się to możliwe dzięki częściowemu włączeniu się autora do prac pielęgnacyjnych. Udało się skoordynować prace i nie dopuścić do powstania różnych wersji kompilatora, które trzeba by było jednocześnie pielęgnować. Oczywiście miało to też ujemne skutki, gdyż efekty moich zmian, często wprowadzanych stopniowo i nie zawsze z powodzeniem, oddziaływały na zwykłych użytkowników.

3. Podstawowe założenia

System Loglan-82 składa się z prekompilatora generującego kod pośredni, oraz interpretera wykonującego ten kod. Sam prekompilator dzieli się fizycznie na dwa przebiegi (w rzeczywistości znacznie więcej) komunikujące się poprzez plik tymczasowy. Te dwie części prekompilatora zwane będą dalej kompilatorem i generatorem (gdyż nazwy te są używane powszechnie w zespole).

Projekt zakładał, że zmiany będą dokonywane głównie w interpreterze napisanym pierwotnie w Pascalu. Brak wygodnej niezależnej kompilacji w połączeniu z nie najlepszą realizacją Pascala byłby bardzo uciążliwy przy przewidywanej dużej ilości zmian. Dlatego też interpreter został przepisany ręcznie na język C, początkowo prawie bez żadnych zmian. Okazało się później, że mimo pozornej bezcelowości krok ten dał wiele korzyści, związanych zresztą nie tylko z realizacją równoległości, m.in. przenośną realizację plików o dostępie bezpośrednim, wykorzystanie całej dostępnej pamięci, przeniesienie interpretera pod system operacyjny XENIX.

Zmiany dokonywane w systemie są zaznaczane w tekstach źródłowych kompilatora i generatora, natomiast w przypadku interpretera były one na tyle liczne, iż nie było to możliwe.

4. Nowe instrukcje L-kodu

L-kod jest niskopoziomową reprezentacją programu, która jest przekazywana od kompilatora do generatora poprzez plik roboczy. Jest to kod trójadresowy zaprojektowany specjalnie dla realizacji

Loglanu na komputerze MERA-400. Argumenty instrukcji L-kodu odwołują się do tablicy symboli kompilatora, więc L-kod nie nadaje się do bezpośredniej interpretacji. Generator GEN zajmuje się m.in. właśnie wyliczaniem adresów argumentów instrukcji. Poza tym dokonuje on jeszcze paru innych modyfikacji (np. zamiany pewnych instrukcji na inne). Kod po przetworzeniu przez generator jest nazywany L'-kodem. W zasadzie zbiór instrukcji L'-kodu jest podzbiorem L-kodu. Różnią się one przede wszystkim postacią argumentów.

Dodano następujące instrukcje L-kodu:

LRESUME	220	wznowienie procesu
LSTOP	221	zatrzymanie procesu
LKILLTEMP	222	zabicie szkieletu procesu
LENABLE	223	odblokowanie procedur
LDISABLE	224	zablokowanie procedur
LACCEPT1	225	początek instrukcji **accept**
LACCEPT2	226	zakończenie instrukcji **accept**
LBACKRPC	227	instrukcja **return**
LASKPROT	228	zapytanie o numer prototypu.

Wszystkie powyższe instrukcje mają swoje odpowiedniki w L'-kodzie z tym, że instrukcje LENABLE, LDISABLE, LACCEPT1 i LBACKRPC mają dodany dodatkowy argument oznaczający długość listy numerów prototypów (punkt 6.1).

5. Zmiany w kompilatorze

Projekt zakładał, że zmiany w kompilatorze będą wprowadzane tylko w razie konieczności. Udało się uniknąć dołączenia nowych globalnych struktur danych lub zmiany istniejących. Wszystkie zmiany w kompilatorze są zaznaczone w tekście źródłowym. W poniższym opisie podane są w nawiasach nazwy plików źródłowych, w których znajdują się omawiane fragmenty kompilatora.

Do niektórych części kompilatora istnieje dokumentacja. Terminologia używana przez autora pochodzi przede wszystkim z tej dokumentacji.

5.1. Analiza leksykalna

Dodano trzy nowe słowa kluczowe w tablicy HASH (HASH.FOR): **enable**, **disable**, **accept**. W procedurze KEY (SCAN.FOR) dodano ich

wejściowe lub wyjściowe). Ostatnie rozróżnienie nie jest uwzględniane przez kompilator (jest natomiast zrealizowane w interpreterze).

5.4. Analiza semantyczna i generowanie L-kodu.

Przede wszystkim trzeba było rozszerzyć obsługę symboli kodu pośredniego przekazywanego przez analizator składniowy w procedurze SDPDA (AL11.FOR) o nowe symbole (aż do 94).

Dla symbolu 52 (instrukcja **resume**), dotychczas ignorowanego, wywołuje się nowa procedura SRESUM (RESUME.FOR). Wykonuje ona te same akcje co analogiczna procedura SATTACH dla instrukcji **attach**, z tym, że generowana jest instrukcja L-kodu LRESUME (działanie opisane w punkcie 7.2.2.3).

Procedura SRETURN (AL11.FOR) wywoływana dla symbolu 53 (**return**) została rozszerzona o generowanie instrukcji L-kodu dla opcji **enable/disable**. L-kod wygląda teraz następująco:

LBACKRPC (adres prototypu)* 0

227

Adres prototypu jest zanegowany jeśli odpowiednia procedura ma być zablokowana (punkt 7.2.3.3).

Dla symbolu 57 (**stop**) generowana jest teraz pojedyncza instrukcja L-kodu LSTOP (7.2.2.3).

Dla symboli 91, 92, 93 (**enable, disable** lub **accept**) wywoływana jest nowa procedura SCONC (RESUME.FOR) z parametrem oznaczającym odpowiednią akcję. Procedura ta generuje następujące instrukcje L-kodu:

(LENABLE | LDISABLE | LACCEPT1) (adres prototypu)* 0

223 224 225

Dodatkowo tylko dla instrukcji **accept** jest generowana instrukcja: LACCEPT2 (226). Jest ona wymagana ze względu na możliwość zawieszenia się procesu podczas wykonywania instrukcji **accept** (patrz 7.2.3.4).

Symbol 94 oznaczający koniec listy identyfikatorów procedur jest ignorowany, gdyż nigdy nie powinien pojawić się bez kontekstu.

Inne zmiany nie są już związane z wprowadzeniem nowych instrukcji języka źródłowego.

W procedurze SCALLE (AL12.FOR), obsługującej powrót z generowanego obiektu, dodano generowanie kodu usuwającego lokalny

szkielet procesu utworzony w pamięci procesu generującego (patrz punkt 7.2.2.1.). Szkielet jest usuwany po powrocie z generacji właściwego procesu. Służy do tego nowa instrukcja L-kodu LKILLTEMP (222).

Wyznaczanie prototypu procedury wirtualnej wołanej w sposób zdalny (przez kropkę) obsługiwane przez procedurę SPRFLD (AL12.FOR) uprzednio korzystało z instrukcji L-kodu LVIRTDOT (44 lub 45). Ponieważ w przypadku wołania procedury wirtualnej w innym procesie konieczna jest komunikacja z tym procesem przed wywołaniem (i co za tym idzie zawieszenie procesu - punkt 7.2.6), trzeba było tę instrukcję rozdzielić na dwie: LASKPROT (228) i LVIRTDOT (44 lub 45).

6. Zmiany w generatorze

6.1. Zmiany związane z dodaniem nowych instrukcji L-kodu

Tablica OPDESCR zawierająca opisy typów argumentów instrukcji L-kodu została rozszerzona o nowe instrukcje. Wartości początkowe nadawane są elementom tej tablicy w procedurze CODE. Zmieniono opisy instrukcji LVIRTDOT (44 i 45) uwzględniając ich rozdzielenie na dwie instrukcje.

W procedurze CODE zadeklarowano stałą MAXPROCLIST i tablicę PROCLIST o długości równej wartości tej stałej. W tablicy tej zapamiętywane są numery prototypów procedur będących parametrami aktualnie przetwarzanej instrukcji zmieniającej maskę procedur.

W procedurze SEGMENTS napotkanie instrukcji zmieniającej maskę procedur (223, 224, 225, 227) powoduje zakończenie bieżącego segmentu i wywołanie nowej procedury MAKEPROCLIST. Zamienia ona adresy prototypów w tablicy IPMEM na numery prototypów, którymi posługuje się system wykonawczy (korzystając z pola -1 w prototypie kompilatora, które jest wypełniane przez generator). Lista numerów prototypów jest wpisywana do tablicy PROCLIST. Zachowana jest przy tym konwencja dla instrukcji **return**: numer zanegowany odpowiada żądaniu zablokowania odpowiedniej procedury. Długość listy prototypów jest zapamiętywana jako pierwszy argument bieżącej czwórki (tzn. ostatniej w tym segmencie).

Procedura GEN generująca instrukcje L'-kodu dla interpretera została odpowiednio rozszerzona. Nowe instrukcje L-kodu 220, 221, 222, 226, 228 są obsługiwane standardowo (przetwarzanie sterowane opisem z tablicy OPDESCR). Natomiast dla instrukcji 223, 224, 225,

227 (są to instrukcje zmieniające maskę procedur) wykonywane są specjalne akcje. Wykorzystywana jest przy tym tablica PROCLIST zawierająca listę numerów prototypów. Generowane są następujące instrukcje L'-kodu:

(223 | 224 | 225 | 227) (numer prototypu)*

Długość listy prototypów jest przekazywana jako jedyny argument instrukcji (tryb adresowania 6). Bezpośrednio po instrukcji występuje ciąg numerów prototypów.

6.2. Zmiany w prototypach systemu wykonawczego

Konieczne było rozszerzenie prototypów o nowe pola. Mają one znaczenie tylko dla niektórych rodzajów modułów, ale występują w każdym prototypie.

Pole MASKBASE dla procesów zawiera najmniejszy numer prototypu procedury zadeklarowanej bezpośrednio w procesie lub jego łancuchu prefiksowym. Pole MASKSIZE zawiera liczbę bajtów potrzebną do zapamiętania maski procedur dla procesu. Powyższe dwa pola zostały wprowadzone żeby przyspieszyć operacje na masce procedur w interpreterze, a także żeby zmniejszyć zużycie pamięci.

Pole VIRTNUMBER dla procedur i funkcji zawiera numer procedury wirtualnej lub -1 jeśli dana procedura nie jest wirtualna.

Istniejące już pole PFDESCR zawierające opisy typów parametrów formalnych procedur i funkcji nabrało znaczenia także dla procesów.

Powyższe zmiany w prototypach należało uwzględnić w części generatora tworzącej prototypy.

Procedurę MAKEPARLIST zmieniono tak, aby opisy typów parametrów generowała także dla procesów. W związku z tym trzeba w procedurze LISTS kopiować pole PFDESCR z prototypu modułu prefiksowanego do prototypu modułu prefiksującego. Poprzednio nie było to potrzebne, gdyż procedury i funkcje nie mogły być prefiksami.

Pole VIRTNUMBER jest inicjowane w procedurze PDESCR według słowa +27 w prototypie kompilatora. Dla procedur niewirtualnych wpisywana jest wartość -1.

Wartości pól MASKBASE i MASKSIZE są obliczane w nowej procedurze RPCMASK wołanej w procedurze GENPROT. Dla uproszczenia w aktualnej realizacji uwzględniany jest maksymalny zakres numerów prototypów (tzn. wszystkie moduły z całego programu). Ze względu

na praktycznie ograniczoną liczbę modułów koszt reprezentacji maski procedur jest i tak praktycznie pomijalny.

6.3. Zmiany związane z obsługą obiektów procesów i procedur

Obiekty procesów istniały już w pierwotnej realizacji, ale ich struktura powodowała, że nie były one specjalnie użyteczne. Obecnie obiekt procesu zawiera następujące pola (oprócz tych, które posiada jako współprogram):

- CHD - zawiera adres wirtualny aktualnie wykonywanego współprogramu (tzw. głowy współprogramu)
- VIRTSC - to pole jest traktowane jak pole robocze zawierające adres wirtualny. Może ono mieć różną interpretację, istotny jest fakt, że pole to jest uaktualniane podczas kompaktyfikacji.
- DISPLAY - pole o długości zależnej od liczby modułów w programie zawierające tablicę adresów fizycznych obiektów aktualnie widocznych
- DISPLAY2 - jak wyżej z tym, że zawiera adresy pośrednie.

Poza tym w każdej procedurze jest dodatkowe pole RPCDL zawierające adres globalny procesu, który wywołał w sposób obcy tę procedurę. Pole to nie ma znaczenia dla obiektów procedur wywołanych w normalny sposób.

Powyższe pola są uwzględniane przez procedurę ENDUNIT obliczającą apetyty obiektów.

Obiekt programu głównego, uprzednio generowany statycznie w generatorze (właściwie nie był to obiekt w sensie algorytmów zarządzania pamięcią w interpreterze), jest teraz normalnym obiektem procesu generowanym dynamicznie.

W związku z tym procedura GENPROT rodzaj modułu programu głównego ustawia na PROCESS a nie BLOCK. W procedurach CODE i GENERATOR usunięto zbędne instrukcje dotyczące programu głównego (m.in. generowanie obiektu MAIN i tablicy DISPLAY).

Wprowadzono nowy tryb adresowania używany przy dostępie do zmiennych w obiektach widocznych. Adres tablicy DISPLAY musi być teraz wyliczany dynamicznie w interpreterze, więc argumentem jest tylko przesunięcie. Adres DISPLAY jest dodawany automatycznie przez część interpretera dekodującą instrukcje L'-kodu. Ten tryb adresowania jest używany także przy dostępie do zmiennych globalnych. Poprzednio używane były stałe adresy ze względu na statyczność obiektu programu głównego. Zmiany związane z nowym

trybem adresowania dotyczyły procedur ARGUMENT i RESULT odpowiedzialnych za generowanie odpowiednich opisów argumentów w instrukcjach L'-kodu.

Warto dodać, że nie wszystkie z opisanych w tym punkcie zmian były konieczne ze względu na przyjętą realizację pamięci dla procesów (punkt 7.1.1). Zmiany te umożliwiają jednak ewentualną realizację procesów dzielących pamięć.

7. Zmiany w interpreterze

W tej części systemu zmiany były największe. Ze względu na ich ilość nie było możliwe zaznaczanie ich w tekście źródłowym interpretera. Nie znaczy to wcale, że interpreter został napisany od nowa, wiele jego fragmentów różnej wielkości pozostało nietkniętych (nie licząc przepisania z Pascala na C). Ponieważ niektórych części systemu wykonawczego (szczególnie dotyczących dynamicznej kontroli typów) w ogóle nie udało się zrozumieć, nie zostały one zmodyfikowane. Stąd wynikają niektóre z ograniczeń opisanych w podręczniku użytkownika.

Realizacja miała pozwalać na wykonywanie programów równoległych zarówno z użyciem sieci jak i bez niej, i to prawie bez żadnych zmian w programie Loglanowym. Dla uproszczenia potraktowano równoległość symulowaną jako szczególny przypadek równoległości rozproszonej (rozróżnienie następowało dopiero na najniższym poziomie). Mimo to dużą trudność sprawiło uniknięcie blokady podczas komunikacji procesów z tego samego komputera. Powodem tego jest brak równoległości w języku programowania, w którym zrealizowany jest interpreter.

Konieczne stało się wprowadzenie specjalnego typu wskaźników do obiektów procesów, mających znaczenie na więcej niż jednym komputerze. Musiały być one przechowywane w normalnych zmiennych wskaźnikowych co powodowało ograniczenia na ich postać. Ostatecznie przyjęto, że globalny wskaźnik do procesu składa się z trzech pól: numeru węzła w sieci (1 bajt), numeru procesu w węźle (1 bajt) oraz licznika (2 bajty) służącego do wykrywania wskaźników do procesów zabitych. Licznik ma takie samo znaczenie jak w przypadku normalnych (lokalnych) adresów wirtualnych z tym, że jest on ujemny. Umożliwia to odróżnienie wskaźnika globalnego od lokalnego.

7.1. Struktury danych

Wszystkie struktury danych wprowadzone specjalnie w celu realizacji proponowanego mechanizmu są zdefiniowane w pliku nagłówkowym PROCESS.H. Oprócz tego plik QUEUE.H zawiera definicje struktur kolejek i stosów oraz operacji na nich.

7.1.1 Pamięć dla procesów

Wszystkie tworzone przez użytkownika procesy, nawet te wykonujące się na jednym komputerze, są zrealizowane jako rozproszone. Każdy proces posiada własną pamięć w postaci liniowej tablicy (pole M w deskryptorze procesu) zawierającej kod programu, statyczne struktury danych systemu wykonawczego, obiekty oraz tablicę adresów wirtualnych. Kod całego programu jest powielony w pamięci każdego procesu gdyż tego wymagał sposób w jaki kod generowany przez istniejący kompilator odwoływał się do pamięci. Jest to poważna nieefektywność, ale jej usunięcie nie było możliwe przy sensownym nakładzie pracy.

7.1.2 Deskryptory procesów

Wiele innych globalnych struktur danych interpretera (poza pamięcią) musiało zostać powielonych dla każdego procesu. Zostały one umieszczone w rekordach zwanych deskryptorami procesów (typ procdescr). Globalna tablica deskryptorów procesów (zmienna process) jest zadeklarowana statycznie, co narzuca ograniczenie na liczbę procesów istniejących na jednym komputerze (aktualnie 64 ale łatwo to zmienić). Rozwiązanie takie zostało przyjęte ponieważ procesy są identyfikowane przez swoje numery odpowiadające indeksom w tej tablicy. Chodziło o to aby identyfikator procesu nie zajmował więcej niż jeden bajt (z powodów wspomnianych wyżej). Nie byłoby to możliwe w przypadku gdyby identyfikator procesu był adresem deskryptora (ewentualnie dostęp do deskryptora byłby bardzo nieefektywny).

7.1.3. Kolejka procesów gotowych

Nie wszystkie procesy istniejące na danym komputerze są w danej chwili wykonywane. Procesy mogą być zawieszone z różnych powodów (np. przez instrukcję **stop**, czekanie na obce wywołanie).

Procesy, które są gotowe do wykonania są elementami globalnej kolejki procesów gotowych (zmienna ready). Zawiera ona identyfikatory procesów (indeksy w tablicy deskryptorów).

Głowa kolejki zawiera proces, który jest aktualnie wykonywany przez procesor. Oprócz tego jest on wskazywany przez zmienne thispix (identyfikator) i thisp (adres deskryptora).

7.1.4. Komunikaty

Komunikaty stanowią jedyny sposób komunikacji między różnymi węzłami w sieci. Ze względu na wspomnianą na wstępie strategię są one używane także do komunikacji pomiędzy procesami istniejącymi na tym samym komputerze.

Komunikat (typ message) zawiera zawsze adres nadawcy. W zależności od typu komunikat może też zawierać dodatkowe informacje. Wszystkie komunikaty oprócz żądania utworzenia procesu zawierają adres procesu, do którego są kierowane. Typy komunikatów i pozostałe informacje od nich zależne są następujące:

ERRSIG - błąd, numer błędu
RESUME - wznowienie procesu, -
CREATE - utworzenie procesu, numer prototypu, parametry
CREACK - powrót z tworzenia procesu, parametry wyjściowe
KILLPR- zabicie procesu, -
RPCALL - obce wywołanie, numer prototypu, parametry wejściowe
RPCACK - powrót z obcego wywołania, parametry wyjściowe
ASKPRO - zapytanie o numer prototypu, -
PROACK - odpowiedź na powyższe zapytanie, numer prototypu

Cały komunikat musi zmieścić się w 80 bajtach (jest to wymagane przez program obsługi sieci). Stąd wynika ograniczenie na sumę apetytów parametrów procesu i procedury wołanej w innym procesie.

7.1.5. Globalna kolejka komunikatów

Każdy komunikat, który został odebrany przez program obsługi sieci, jest umieszczany w globalnej kolejce (zmienna globmsgqueue).

7.1.6. Lokalne kolejki komunikatów

Niektóre komunikaty zostają obsłużone od razu, natomiast inne

wymagają, aby dotyczący ich proces był w trakcie swojego kwantu czasu. Takie komunikaty są przenoszone z kolejki globalnej do kolejki lokalnej związanej z danym procesem (pole msgqueue w deskryptorze procesu).

7.1.7. Kolejki procesów czekających na obsługę obcego wywołania

Z każdym procesem jest związana lokalna kolejka procesów czekających na obsługę obcego wywołania procedury w tym procesie (pole rpcwait w deskryptorze procesu). Właściwie jest to kolejka komunikatów wysłanych przez te procesy. Oprócz adresu procesu wołającego zawierają one numer prototypu procedury wołanej oraz przekazane parametry wejściowe.

7.1.8. Stos masek procedur

Dla każdego procesu pamiętana jest maska odblokowanych procedur. Zgodnie z semantyką obcego wołania jest ona zerowana w momencie wejścia do procedury i odtwarzana po jej zakończeniu. Wykonywanie procedury może zostać przerwane przez inne obce wołanie (jeśli maska zostanie zmieniona). Powoduje to konieczność pamiętania w deskryptorze procesu stosu masek procedur (pole rpcmask). Aktualna maska znajduje się na szczycie tego stosu.

7.1.9. Zmiany w obiektach i prototypach

Zostały one już omówione w punktach 6.2. i 6.3.

7.2. Algorytmy

7.2.1. Inicjowanie interpretera

W tej części zaszły duże zmiany w stosunku do wersji sekwencyjnej. Jest to jedyna faza, w której działanie interpretera zależy od tego czy jest on uruchomiony na konsoli czy na innym komputerze (patrz podręcznik użytkownika). Sytuacja ta jest rozpoznawana za pomocą opcji /r podawanej przez użytkownika przy wywołaniu interpretera.

Interpreter zaczyna działanie od ustalenia konfiguracji, tzn. sprawdzenia czy jest dostępna sieć, czy jest to interpreter odległy (tzn. uruchomiony nie na konsoli) oraz jaki będzie

przydział pamięci dla każdego procesu.

Następnie interpreter alokuje tablicę pamięci dla jednego procesu i do tej tablicy wczytuje L'-kod z pliku podanego przez użytkownika jako parametr wywołania interpretera. Z innego pliku wczytywane są prototypy systemu wykonawczego.

Inicjowanie systemu wykonawczego polega na wstępnym wypełnieniu wszystkich deskryptorów procesów, obliczeniu względnych adresów tablic DISPLAY (zależnych od liczby modułów w programie) oraz utworzeniu pustej kolejki procesów gotowych.

Jeśli interpreter został wywołany na konsoli to tworzy od razu proces programu głównego i wstawia go do kolejki procesów gotowych.

Na zakończenie tego etapu zostają odblokowane przerwania generowane przez zmodyfikowany program obsługi sieci (punkt 7.2.5.2).

7.2.2. Zarządzanie procesami

7.2.2.1. Rozpoczęcie generacji procesu

Każdy proces oprócz programu głównego jest tworzony przez inny proces za pomocą generatora **new**. Żeby nie zmieniać w zasadniczy sposób kodu generowanego przez kompilator dla tej konstrukcji, przyjęto poniższe rozwiązanie.

Generacja obiektu procesu przebiega normalnie (tzn. tak jak np. klasy) aż do momentu przekazania sterowania do tego obiektu. Oznacza to, że najpierw w pamięci procesu generującego tworzony jest tymczasowy obiekt procesu (zwany dalej szkieletem). Parametry wejściowe wpisywane są do tego szkieletu tak jak w przypadku innych modułów.

W momencie przekazania sterowania system wykonawczy odróżnia ten przypadek jako specjalny (procedura go()). Adres wirtualny szkieletu procesu zapamiętywany jest w deskryptorze procesu generującego. Tworzony jest komunikat żądający utworzenia procesu (typu CREATE) zawierający wszystkie parametry wejściowe odczytane ze szkieletu. Komunikat ten wysyłany jest do komputera, na którym ma być utworzony nowy proces. Następnie proces generujący zawiesza się w oczekiwaniu na powrót z obiektu generowanego.

Komunikat żądający utworzenia procesu zostaje odebrany przez interpreter działający na docelowym komputerze. Jego obsługą zajmuje się procedura createprocess(). Najpierw wyszukuje ona nie

używany deskryptor procesu. Przydzielana jest pamięć dla nowego procesu, jeśli nie została przydzielona wcześniej dla innego procesu korzystającego z tego samego deskryptora. Jeśli trzeba było przydzielić pamięć, to jej statyczna część (kod i niektóre struktury danych systemu wykonawczego) jest inicjowana poprzez przepisanie odpowiedniego obszaru z pamięci dla procesu o numerze 0 (przydzielonej i wypełnionej na początku działania interpretera). Następnie wypełniany jest deskryptor procesu (procedura initprocess()).

Wartości parametrów wejściowych procesu są ustalane na podstawie komunikatu otrzymanego od procesu generującego. Utworzony proces jest aktywowany (wstawiany do kolejki procesów gotowych) i system wykonawczy oddaje sterowanie przejęte na czas obsługi komunikatu.

7.2.2.2. Powrót z generacji procesu

System wykonawczy traktuje w specjalny sposób również wykonanie pierwszej instrucji **return** w procesie (procedura back()). Jeśli jest ona wykonywana przez program główny to proces programu głównego kończy się (ale niekoniecznie cały program). W przeciwnym przypadku tworzony jest komunikat oznaczający powrót z generacji procesu zawierający adres globalny nowego procesu (jako adres nadawcy) oraz parametry wyjściowe odczytane z obiektu procesu. Komunikat ten wysyła się do procesu generującego, którego adres był zapamiętany w polu DL obiektu procesu. Jednocześnie pole to jest zerowane, aby zapewnić prawidłową obsługę sygnałów zgłaszanych w procesie.

Interpreter wykonujący proces generujący po odebraniu komunikatu przepisuje parametry wyjściowe do szkieletu nowego procesu. Adres globalny procesu jest zapamiętywany w specjalnym polu deskryptora procesu (backobj). Następnie interpreter wznawia proces generujący.

Proces generujący już bez udziału systemu wykonawczego odczytuje adres nowego procesu oraz parametry wyjściowe tak jakby szkielet procesu był tym nowym procesem, a następnie usuwa szkielet procesu generowanego. W tym właśnie celu konieczne było wprowadzenie dodatkowej instrukcji L-kodu LKILLTEMP, gdyż normalnie obiekty procesów (w ogólności klas) nie były zabijane po powrocie.

7.2.2.3. Zatrzymywanie i wznawianie procesu

Operacje te występują w dwóch postaciach. Po pierwsze odpowiadają one instrukcjom **stop** i **resume** z języka. Poza tym system wykonawczy korzysta z wewnętrznych operacji o podobnym działaniu.

W przypadku instrukcji **stop** różnica ta jest mniej uchwytna, gdyż zatrzymanie procesu może nastąpić tylko na skutek akcji wykonanej przez niego samego. Instrukcja **stop** jest tłumaczona na pojedynczą instrukcję L-kodu LSTOP. Jej obsługa w interpreterze polega na bezpośrednim wywołaniu procedury systemu wykonawczego passivate(). Procedura ta usuwa po prostu proces z kolejki procesów gotowych.

Wznowienie procesu jest operacją bardziej skomplikowaną, gdyż dokonywane jest przez inny proces. Obsługa instrukcji **resume** (procedura resume()) polega na wysłaniu komunikatu żądającego wznowienia procesu do komputera, na którym istnieje ten proces.

Interpreter odbierający po sprawdzeniu poprawności wywołuje operację systemu wykonawczego activate(). Procedura ta wstawia odpowiedni proces do kolejki procesów gotowych.

7.2.2.4. Zakończenie procesu

Zakończenie procesu następuje po wykonaniu wszystkich instrukcji w treści procesu lub na skutek nie znalezienia modułu obsługi wyjątków dla sygnału. Ponieważ inne procesy nie mają żadnego dostępu do zasobów czy atrybutów procesu jeśli jest on nieaktywny (a już nigdy nie będzie) to proces zakończony jest usuwany (patrz niżej). Po zakończeniu (i usunięciu) ostatniego procesu na danym komputerze interpreter kończy działanie. Dzięki temu programy sekwencyjne zachowują dotychczasową semantykę. Po zakończeniu interpretera na jednym komputerze mogą jednak działać jeszcze procesy wykonywane na innych komputerach.

7.2.2.5. Usuwanie procesu

Wykonanie instrukcji **kill** żądającej usunięcia procesu (procedura gkill()) polega na wysłaniu komunikatu do odpowiedniego interpretera. Po jego odebraniu system wykonawczy zwalnia pamięć zajmowaną dotychczas przez lokalne kolejki procesu oraz stos masek procedur. Deskryptor procesu zaznacza jako nie używany. Nie

zwalnia natomiast tablicy symulującej pamięć procesu. Może ona zostać wykorzystana powtórnie bez żadnych zmian i strat (patrz punkt 7.2.2.1.).

7.2.2.6. Podział czasu

Jak już wspomniano wyżej, podział czasu pomiędzy symulowane procesy jest oparty o kolejkę procesów gotowych. Po wykonaniu każdej instrukcji L'-kodu jest wywoływana procedura schedule(), która podejmuje decyzję o zmianie kontekstu. Proces bieżący jest zmieniany jeśli wykorzystał on swój kwant czasu lub podczas wykonywania ostatniej instrukcji była modyfikowana kolejka procesów gotowych. Jeśli spełniony jest któryś z tych warunków, proces bieżący jest usuwany z kolejki i wstawiany na jej koniec. Nowym procesem bieżącym jest proces, który jest teraz na początku kolejki.

Jeśli po wykonaniu instrukcji L'-kodu kolejka procesów gotowych jest pusta to system wykonawczy oczekuje w pętli na nadejście jakiegoś komunikatu (i obsługuje go jeśli przyjdzie) tak długo jak długo kolejka pozostaje pusta.

Właściwa zmiana kontekstu (procedura transfer()) polega na zapamiętaniu istotnych informacji (licznik rozkazów i adresy obiektu bieżącego) w deskryptorze byłego procesu bieżącego i odtworzeniu ich z deskryptora nowego procesu bieżącego.

Odliczanie kwantów czasu dla procesów jest oparte o czas rzeczywisty. Pod systemem PC-DOS korzysta się z zegara obsługiwanego przez BIOS (bajt pod adresem 0040:006C). W przypadku systemu XENIX wywoływana jest funkcja biblioteczna clock(). Jest to jedyna część interpretera różniąca się w przypadku tych dwóch systemów.

Należy zaznaczyć, że wywłaszczanie procesów nie zwalniających dobrowolnie procesora nie jest całkowicie pełne. Proces, który zawiesi się w oczekiwaniu na operację wejścia/wyjścia (np. czytanie z klawiatury lub pisanie na drukarkę) tym samym zawiesza cały interpreter i wszystkie procesy wykonywane przez ten interpreter. Wynika to bezpośrednio z jednoprogramowości systemu operacyjnego i nie daje się obejść w sensowny sposób.

7.2.3. Obce wywołanie procedury

7.2.3.1. Wysłanie i przyjęcie żądania

Podobnie jak w przypadku tworzenia procesu obiekt procedury jest tworzony najpierw w pamięci procesu wołającego (obiekt ten zwany jest dalej szkieletem). System wykonawczy odróżnia obce wywołanie od normalnego dopiero w momencie przekazania sterowania do wygenerowanego szkieletu (procedura go()). Tworzy wtedy odpowiedni komunikat (typu RPCALL) zawierający numer prototypu wołanej procedury i jej parametry wejściowe odczytane ze szkieletu. Komunikat ten wysyła do procesu wołanego i zawiesza proces wołający w oczekiwaniu na zakończenie wołania.

Interpreter odbierający żądanie obcego wywołania procedury nie obsługuje go bezpośrednio ale kieruje do odpowiedniej kolejki. Zajmuje się tym procedura rpc1(). Jeśli dana procedura jest zablokowana w procesie wołanym, to komunikat jest wstawiany do kolejki komunikatów od procesów oczekujących na obsługę obcego wywołania (punkt 7.1.7), w przeciwnym przypadku natomiast do lokalnej kolejki komunikatów. Jeśli procedura jest odblokowana a proces wołany był zawieszony podczas wykonywania instrukcji **accept** to jest on wznawiany.

7.2.3.2. Rozpoczęcie wykonania procedury

Przerwanie procesu w celu wykonania procedury może nastąpić tylko pomiędzy instrukcjami języka źródłowego, gdyż przerwanie instrukcji stwarza możliwość, że tymczasowe adresy fizyczne obiektów staną się nieaktualne. Dlatego przy każdej instrukcji L'-kodu LTRACE sprawdzane jest (procedura rpc2()) czy w lokalnej kolejce komunikatów znajduje się żądanie wywołania odblokowanej procedury (pozostałe żądania są przenoszone do kolejki procesów czekających na obsługę obcego wywołania - funkcja rpcready()). Jeśli takie żądanie zostało znalezione to wywoływana jest procedura rpc3(), której zadaniem jest rozpoczęcie wykonania procedury.

Komunikat żądający obcego wywołania odblokowanej procedury jest usuwany z lokalnej kolejki. Wszystkie procedury zostają zablokowane w bieżącym procesie poprzez włożenie pustej maski na stos masek procedur. Następnie otwierany jest obiekt wołanej procedury. Pole RPCDL w tym obiekcie jest ustawiane tak aby wskazywało na proces wołający. Parametry wejściowe procedury zawarte w komunikacie są wpisywane w odpowiednie miejsca w obiekcie. Wtedy dopiero sterowanie jest przekazywane do obiektu

procedury (przez wywołanie procedury go()).

7.2.3.3. Zakończenie obcego wywołania

Zakończenie obcego wywołania rozpoznaje procedura back() po zawartości pola RPCDL w obiekcie procedury. Najpierw wykonuje ona standardowe akcje związane z powrotem z obiektu (m.in. uaktualniane są tablice DISPLAY i DISPLAY2 oraz wskaźnik obiektu bieżącego w procesie, sterowanie jest przekazywane do poprzednika dynamicznego). Dopiero po tym wysyła do procesu wołającego komunikat informujący o zakończeniu obcego wywołania i zawierający parametry wyjściowe procedury przepisane z obiektu. Na koniec obiekt procedury jest usuwany. W tym miejscu złamano zasadę, że usuwanie obiektów jest wykonywane przez instrukcje L'-kodu, a nie przy okazji innych akcji systemu wykonawczego. Następnie maska procedur w bieżącym procesie jest odtwarzana do stanu sprzed rozpoczęcia wywołania. Jeśli zakończenie procedury nastąpiło na skutek instrukcji **return** z opcjami **enable/disable** to maska jest jeszcze odpowiednio modyfikowana (patrz 7.2.4. i 7.2.5.).

Po stronie wołającej, po odebraniu komunikatu o zakończeniu obcego wywołania, system wykonawczy przepisuje parametry wyjściowe do szkieletu procedury i wznawia proces wołający. Dalsze akcje wykonywane przez proces wołający są takie same jak po każdym powrocie z procedury, więc żadne zmiany nie były potrzebne.

7.2.3.4. Instrukcja **accept**

Jak już wspomniano kod dla instrukcji **accept** składa się z dwóch instrukcji L'-kodu. Pierwsza to LACCEPT1. Jej jedynym argumentem jest długość listy procedur, które mają być odblokowane. Lista numerów prototypów znajduje się zawsze zaraz za instrukcją w kodzie programu. Bezpośrednio za nią następuje bezargumentowa instrukcja LACCEPT2.

Rozpoczęcie wykonywania instrukcji **accept** (instrukcja L'-kodu LACCEPT1) jest zadaniem procedury accept(). Wierzchołek stosu masek procedur (tzn. aktualna maska) jest duplikowany. Nowa bieżąca maska, początkowo taka sama jak poprzednia, jest modyfikowana poprzez odblokowanie procedur, których identyfikatory pojawiły się w instrukcji **accept**. Następnie jeśli w lokalnej kolejce komunikatów nie ma żądania wywołania którejś z odblokowanych aktualnie procedur, to proces jest zawieszany.

Przejście do wykonania kolejnej instrukcji (tzn. LACCEPT2) następuje albo jeśli proces nie zawiesi się w instrukcji LACCEPT1 albo na skutek wznowienia procesu przez komunikat żądający wywołania odblokowanej procedury (patrz 7.2.3.1). W obu przypadkach jest zagwarantowane, że pierwszy element lokalnej kolejki komunikatów zawiera właściwe żądanie. Wykonanie instrukcji LACCEPT2 polega na odtworzeniu maski procedur ze stosu (jest to maska sprzed rozpoczęcia wykonywania instrukcji **accept**) i przejściu do wykonania procedury (procedura rpc3()). Dzięki temu, że maska procedur jest zapamiętywana i odtwarzana przed faktycznym wywołaniem procedury ma ona właściwą zawartość podczas oczekiwania a jednocześnie powrót z procedury odtworzy i zmodyfikuje maskę nie zmienioną przez odblokowanie żądanych procedur.

7.2.4. Operacje na maskach procedur

Maska procedur jest pamiętana jako wektor bitowy indeksowany numerami prototypów procedur. Indeksy są przesunięte zgodnie z zawartością pola MASKBASE w prototypie procesu obliczonego przez generator. Długość maski w bajtach jest wartością innego pola MASKSIZE. Pomimo uproszczonego algorytmu obliczania tych pól stosowanego obecnie przez generator reprezentacja dla tej struktury jest bardzo oszczędna. Nawet dla dużych programów pojedyncza maska zajmuje tylko parę słów maszynowych.

Dostęp do maski realizowany jest przez procedurę bitaccess() obliczającą numer bajtu w masce i wzorzec bitowy dla procedury o podanym numerze prototypu. Korzystają z tej operacji procedury enable(), disable() oraz isenabled() ustawiające, zerujące i testujące odpowiednie bity w masce.

Dodatkowo przewidziane są operacje na stosie masek procedur. Procedury pushmask(), dupmask() i popmask() powodują odpowiednio położenie pustej maski na stos (czyli zablokowanie wszystkich procedur), zduplikowanie wierzchołka stosu (wykorzystywana przez obsługę instrukcji **accept**) oraz odtworzenie poprzedniej maski ze stosu.

7.2.5. Blokowanie i odblokowywanie procedur

Operacje te odpowiadają prawie bezpośrednio operacjom na aktualnej masce procedur (patrz wyżej). Należało jednak uwzględnić dwie okoliczności.

Po pierwsze jeśli procedura, która ma być odblokowana lub zablokowana, jest procedurą wirtualną to jej numer prototypu przekazany przez kompilator (statycznie wyliczony) nie musi wcale odpowiadać tej procedurze, o którą naprawdę chodzi. Z tego względu przed każdą operacją na masce procedur dotyczącą jakiejś procedury sprawdzane jest czy jest to procedura wirtualna. Jeśli tak to odczytywany jest jej tzw. numer procedury wirtualnej i z tablicy będącej częścią prototypu procesu odczytywany jest faktyczny numer prototypu. Zależy on oczywiście od dynamicznego typu obiektu procesu. Sprawdzeniem tym zajmuje się funkcja virtprot().

Poza tym każda operacja, która odblokowuje którąś z procedur, może spowodować, że pewne żądanie czekające w kolejce procesów oczekujących na obsługę obcego wywołania może być teraz obsłużone. Dlatego też po każdej takiej operacji wywoływana jest procedura evaluaterpc(), której zadaniem jest sprawdzenie czy zaszło powyższe zdarzenie. Kolejka procesów oczekujących jest przeglądana w poszukiwaniu pierwszego żądania wywołania procedury aktualnie odblokowanej. Jeśli takie żądanie zostanie znalezione to jest ono usuwane z tej kolejki i wstawiane na początek lokalnej kolejki komunikatów (przed żądania jeszcze nie rozpatrzone).

7.2.6. Procedury wirtualne

Wirtualność procedur ma wpływ na działanie opisywanego mechanizmu w dwóch przypadkach. Przypadek pierwszy to opisane wyżej zablokowanie lub odblokowanie procedury wirtualnej. Tutaj zostanie opisany przypadek drugi to znaczy obce wołanie procedury wirtualnej.

Jeśliby chcieć zachować opisany wyżej mechanizm obcego wywołania to proces wołający musiałby przesłać do procesu wołanego właściwy numer prototypu procedury wołanej, który zależy od dynamicznego typu procesu wołanego. Narzucające się rozwiązanie polegałoby na tym, że przesyłany byłby numer prototypu wyliczony statycznie i zamieniany w procesie wołanym na numer właściwy (np. za pomocą opisanej wyżej funkcji virtprot()). Przyjęto jednak inne rozwiązanie umożliwiające ewentualną realizację dynamicznej kontroli typów na wskaźnikach do procesów (obecnie nie zrealizowaną).

Wprowadzono specjalną operację zapytania odległego procesu o jego typ dynamiczny (tzn. faktyczny numer prototypu). Służy do tego instrukcja L'-kodu LASKPROT. Jej wykonanie polega na wysłaniu

odpowiedniego komunikatu zapytującego i zawieszeniu procesu w oczekiwaniu na odpowiedź. Po stronie odbierającej obsługa jest bardzo prosta. Wystarczy wysłać komunikat powrotny z właściwym numerem prototypu. Proces zapytujący jest wznawiany po uprzednim wpisaniu numeru prototypu w ustalone miejsce w pamięci procesu. Oczywiście w przypadku obiektów nie będących procesami instrukcja LASKPROT wyznacza prototyp bezpośrednio z pola PROTNUM w obiekcie bez zawieszania procesu wykonującego tę instrukcję.

Instrukcja L'-kodu LVIRTDOT wyznaczająca faktyczny prototyp procedury wirtualnej została rozdzielona na dwie. Najpierw wyznacza się numer prototypu obiektu, do którego ona należy korzystając z instrukcji LASKPROT. Następnie właściwa instrukcja LVIRTDOT wyznacza prototyp procedury wirtualnej korzystając z obliczonego wcześniej prototypu obiektu.

7.2.7. Obsługa błędów

Specjalnej obsługi wymagają jedynie błędy wykryte przez interpreter inny niż ten wykonujący proces, który je spowodował. W takim przypadku przesyłany jest odpowiedni komunikat o błędzie do procesu sprawcy. Jeśli proces ten był zawieszony nie za pomocą instrukcji **stop** ale przez system wykonawczy (np. z powodu oczekiwania na obce wywołanie) to jest on wznawiany i wymuszane jest natychmiastowe oddanie mu procesora. Następnie podejmowane są normalne akcje obsługi wyjątków systemowych. Proces zatrzymany przez użytkownika ignoruje nadchodzące komunikaty o błędach.

7.2.8. Kolejki i stosy

Wszystkie kolejki i stosy z wyjątkiem globalnej kolejki komunikatów są obsługiwane przez jeden moduł zawarty w pliku QUEUE.C. Zarówno kolejki jak i stosy są potraktowane jako szczególne przypadki struktury danych zwanej czasami kolejką półtorastronną (ang. semi-deque). Jest ona zrealizowana za pomocą jednokierunkowych list cyklicznych. Kolejka (stos) jest reprezentowana przez wskaźnik do ostatniego elementu (dna stosu). Cykliczność gwarantuje nam, że następny w liście jest pierwszy element w kolejce (wierzchołek stosu). Dzięki temu wszystkie podstawowe operacje mają koszt stały.

Dodatkowo zrealizowane są operacje wyszukania i usunięcia dowolnego elementu z kolejki oraz operacja rotacji (tzn. usunięcia

pierwszego elementu i wstawienia na koniec). Ta ostatnia operacja jest co prawda nadmiarowa ale wykonywana jest bardzo często, więc została wprowadzona w celu zwiększenia efektywności (jej realizacja nie wymaga wywoływania procedur zarządzania pamięcią). Poza tym zrealizowana jest operacja zwalniania pamięci zajmowanej przez kolejkę (stos) i jej elementy.

Globalna kolejka komunikatów (7.1.5.) nie mogła być zrealizowana przy użyciu tego modułu, gdyż operacje na niej są wywoływane przez procedurę obsługi przerwania. Nie mogą one zatem korzystać z procedur dynamicznego przydziału pamięci ze względu na to, że procedury te nie są wielowejściowe (ang. reentrant). Globalna kolejka komunikatów jest zapamiętana w statycznej tablicy traktowanej jako bufor cykliczny. Ze względu na ograniczoną ilość instrukcji jaką można wykonać podczas obsługi przerwania, operacje na tej kolejce są wykonywane bezpośrednio przez manipulację wskaźnikami do początku i końca kolejki (zresztą każda z operacji wstawienia i usunięcia elementu występuje tylko raz).

7.2.9. Współpraca ze sprzętem i oprogramowaniem systemowym

Była to jedna z trudniejszych części pracy ze względu na brak jakiejkolwiek dokumentacji do karty sieciowej i do programu obsługi (ang. driver). W tych warunkach samodzielne programowanie sprzętu absolutnie nie wchodziło w rachubę. Jedynym sposobem było użycie istniejącego programu obsługi i zdobycie wiedzy przez dekompilację jego kodu binarnego i analizę otrzymanego w ten sposób tekstu "źródłowego" (ok. 220KB tekstu w języku maszynowym z adresami absolutnymi i bez słowa komentarza). To co udało się ustalić na pewno nie wyczerpuje wszystkich możliwości sprzętu i programu obsługi. Z pewnością możliwa jest znacznie lepsza realizacja.

Przecięcie wiedzy dostępnej i użytecznej na temat możliwości sieci streszcza się mniej więcej tak. Każdy komputer włączony do sieci (węzeł) jest opatrzony unikalnym numerem (liczbą z przedziału 0..254) zwanym dalej numerem węzła. Węzeł może wysłać komunikat do innego węzła podając jego numer lub do wszystkich węzłów na raz. Ta ostatnia możliwość, choć kuszącą, jest jednak nie do wykorzystania, gdyż interferowałaby z pracą komputerów włączonych do sieci ale nie związanych z wykonującym się programem w Loglanie. Komunikat ma stałą długość 80 bajtów (stąd ograniczenie na liczbę parametrów procesu). Komunikat jest zawsze

odbierany bezpośrednio przez program obsługi i natychmiast wyświetlany na ekranie monitora (!).

Szczególny przypadek zachodzi gdy węzeł znajduje się w stanie ignorującym wszelkie nadchodzące komunikaty. Próba wysłania komunikatu do takiego węzła kończy się zawsze błędem. Przejście w stan ignorujący, jak również wyjście z niego można spowodować programowo.

7.2.9.1. Wysłanie komunikatu

Jest to operacja stosunkowo prosta. Przed wysłaniem komunikat jest uzupełniany zawsze o adres nadawcy. Jeśli jest to komunikat do odległego komputera to wysłanie go jest powierzane programowi obsługi sieci. W przeciwnym przypadku symulowane jest przerwanie poprzez wywołanie procedury obsługi przerwania msginterrupt().

7.2.9.2. Odbiór komunikatu

Na szczęście udało się opracować metodę dynamicznej modyfikacji fragmentu kodu programu obsługi tak aby zamiast wyświetlania komunikatu wywoływana była wskazana procedura. Procedura ta jako parametr ma adres komunikatu, którego nadejście spowodowało przerwanie. Po zakończeniu działania interpretera kod programu obsługi jest odtwarzany do poprzedniego stanu, tak aby umożliwić normalną pracę. W tym celu konieczne jest przechwytywanie przerwania systemowego związanego z naciśnięciem klawisza Control-Break na klawiaturze.

W interpreterze obsługą przerwania związanego z nadejściem komunikatu zajmuje się procedura msginterrupt(). Umieszcza ona po prostu komunikat będący parametrem w globalnej kolejce komunikatów. Jeśli kolejka zapełniła się to węzeł przechodzi w stan ignorujący. Komunikat zostanie usunięty z globalnej kolejki przez procedurę trapmsg() wywoływaną przy każdej instrukcji LTRACE (ewentualnie nastąpi też wyjście ze stanu ignorującego).

ROZDZIAŁ VI
Docelowa realizacja.

Choć dalszy rozwój tej realizacji obcego wołania procedur nie jest przewidywany, w niniejszym rozdziale opisane zostaną te braki realizacji, które powinny być usunięte jeśli miałaby ona mieć szerszy zakres zastosowań.

1. Bezpieczeństwo

Aktualnie żadne z ograniczeń wymienionych w podręczniku użytkownika nie jest sprawdzane przez kompilator. Przeczy to w oczywisty sposób założeniom Loglanu, wśród których jest postulat całkowitego bezpieczeństwa języka. W docelowej realizacji kompilator powinien kontrolować czy program spełnia podane ograniczenia. Niektóre z nich w ogóle nie są możliwe do sprawdzenia w czasie kompilacji, co sugeruje aby je zastąpić innymi (bardziej ograniczającymi, za to statycznymi, np. składniowymi). Możliwe jest też inne podejście do koncepcji rozpraszania procesów.

2. Inna koncepcja rozpraszania procesów

Dobrze byłoby zachować możliwość korzystania z obu rodzajów procesów: rozproszonych i mających wspólną pamięć. Kompilator nie mogąc sprawdzić wprawdzie czy podane ograniczenia na procesy rozproszone są spełnione, może stwierdzić czy istnieje potencjalna możliwość ich przekroczenia. Jeśli tak, to kompilator wymuszałby wykonywanie tych procesów na jednym komputerze. W przeciwnym przypadku programista decydowałby o ich rozproszeniu.

3. Inne operacje na wskaźnikach do procesów

Brak wielu operacji na wskaźnikach do procesów jest związany nie tyle z niemożliwością ich zrealizowania w systemie rozproszonym co raczej ze słabością obecnej realizacji. Do operacji tych należą **is**, **in**, **qua**, porównania oraz dynamiczna kontrola typów. Ich realizacja wymagałaby znacznych zmian w generowanym kodzie ale jest możliwa.

ROZDZIAŁ VII

Ewentualne rozszerzenia obcego wołania procedur.

Obce wołanie procedur jest wystarczającym samodzielnym mechanizmem synchronizacji. Wydaje się jednak, że jego użyteczność wzrosłaby, gdyby rozszerzyć go o pewne dodatkowe konstrukcje. Zostały one tak dobrane, aby dały się zrealizować bez zmiany sposobu realizacji istniejących konstrukcji.

1. Badanie liczby procesów oczekujących

Proces może dowiedzieć się ile procesów oczekuje na obsługę obcego wołania procedury P za pomocą operacji standardowej:

pending(P)

zwracającej liczbę tych procesów. W obecnej realizacji daje się ona wykonać bez trudu - wystarczy przejrzeć kolejkę procesów oczekujących na obsługę obcego wywołania. Największą trudnością jest natomiast wprowadzenie odpowiednich zmian do kompilatora (analiza składniowa i generowanie kodu).

2. Wywołanie asynchroniczne

Instrukcja:

send X.P(...)

działa tak jak instrukcja:

call X.P(...)

to znaczy powoduje obce wywołanie procedury P w procesie X, ale w odróżnieniu od tej ostatniej nie zawiesza procesu wołającego w oczekiwaniu na zakończenie wywołania. Procedura wołana w ten sposób może mieć wtedy tylko parametry wejściowe.

Rozszerzenie o instrukcję **send** upodabnia obce wołanie do mechanizmu występującego w Synchronizing Resources [FF84]. Asynchronicznie wysyłane komunikaty można traktować jako szczególny przypadek takiej wersji obcego wołania procedur. Programista ma do swojej dyspozycji zarówno mechanizmy synchroniczne jak i asynchroniczne tworzące spójną całość.

Jak łatwo zauważyć, dodanie instrukcji **send** do istniejącej realizacji nie sprawiłoby żadnych kłopotów pod warunkiem, że umie się zmodyfikować analizator składniowy i generator kodu.

ROZDIAŁ VIII
Podsumowanie

Wydaje się, że cele postawione w rozdziale I zostały osiągnięte, choć nie bez pewnych zastrzeżeń. Obce wołanie procedur jest pełnowartościowym mechanizmem synchronizacji procesów rozproszonych, szczególnie po dodaniu rozszerzeń opisanych w rozdziale VII. Jak wykazują przykłady i porównanie z innymi mechanizmami, obce wołanie procedur pozwala na stosunkowo wygodne rozwiązywanie wielu problemów synchronizacyjnych. Braki dotyczą przede wszystkim możliwości programowania systemów pracujących w czasie rzeczywistym.

Proponowany mechanizm został dosyć dobrze dopasowany do istniejących konstrukcji języka, czego nie można powiedzieć na przykład o mechaniźmie opierającym się na jawnych komunikatach. Nie twierdzę, że obce wołanie procedur powinno zostać przyjęte jako mechanizm synchronizacji w nowej wersji Loglanu. Uważam jednak, że istniejące obecnie połączenie semaforów binarnych i przerwań należy zastąpić przez jeden spójny i dostatecznie silny mechanizm.

Udało się także dopasować realizację proponowanego mechanizmu do istniejącego dużego systemu. Dzięki temu programista oprócz konstrukcji równoległych ma do swojej dyspozycji bogaty język programowania.

Prototypowa realizacja obcego wołania procedur pozostawia mimo wszystko pewien niedosyt. Było jasne od samego początku, że nie uda się uzyskać produktu w pełni użytkowego. Autor żywił jednak cichą nadzieję, że to co powstanie, będzie mogło służyć chociaż do celów dydaktycznych (jest to jedna z nielicznych realizacji współbieżnego języka programowania dostępnych w Instytucie). W tej chwili wydaje się, że nawet takie zastosowanie może być trudne ze względu na brak gwarancji bezpieczeństwa ze strony systemu. Uzyskanie w pełni niezawodnego i odpornego produktu nie może opierać się na istniejącej, wielokrotnie przenoszonej i modyfikowanej realizacji. Z kolei realizacja języka tak dużego jak Loglan przekracza z pewnością możliwości pojedynczego magistranta.

[BH75] Brinch Hansen P., "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, vol. SE-1, no.2 (June 1975), s. 199-207

[BH78] Brinch Hansen P., "Distributed Processes: A Concurrent Programming Concept", CACM, vol. 21, no. 11 (November 1978), s. 934-941

[BH87] Brinch Hansen P., "Joyce - A Programming Language for Distributed Systems", Software - Practice and Experience, vol. 17, no. 1 (1987), s. 29-55

[BD87] Budkowski S., Dembiński P., "An Introduction to Estelle: A Specification Language for Distributed Systems", Czwarta Jesienna Szkoła PTI, Mrągowo, grudzień 1987

[D83] US Department of Defense, "Ada Programming Language", Military Standard, ANSI/MIL-STD-1815A, Ada Joint Program Office, 1983

[EH87] Elizabeth M., Hull C., "Occam - A Programming Language for Multiprocessor Systems", Computer Languages, vol. 12, no. 1 (1987), s. 27-37

[F79] Feldman J.A., "High Level Programming for Distributed Computing", CACM, vol. 22, no. 6 (June 1979), s. 353-368

[FF84] Filman R.E., Friedman D.P., "Coordinated Computing: Tools and Techniques for Distributed Software", McGraw-Hill, 1984

[G87] Gammage N.D., Kamel R.F., Casey L.M., "Remote Rendezvous", Software - Practice and Experience, vol. 17, no. 10 (October 1987), s. 741-755.

[H78] Hoare C.A.R., "Communicating Sequential Processes", CACM, vol. 21, no. 8 (August 1978), s. 666-677

[IM82] Iszkowski W., Maniecki M., "Programowanie Współbieżne", WNT, Warszawa 1982

[KP84] Kernighan B.W., Pike R., "The UNIX Programming Environment", Prentice Hall, Englewood Cliffs, N.J., 1984

[L87] Kreczmar A., Lao M., Litwiniuk A., Przytycka T., Salwicki A., Warpechowska J., Warpechowski M., Szałas A., Szczepańska-Wasersztrum D., "Report on The Programming Language LOGLAN", IIUW, Warszawa, November 1987 (w przygotowaniu)

[P84] Petermann U., Szałas A., "On Distributed Processes Communicating by Interrupting Signals", IIUW reports, no.

137, 1984

[R84] Recommendation Z.200 (CCITT High Level Language CHILL), CCITT AP VII - No. 21-E, UIT, Geneva 1984

[R79] DEC, "RSX-11M/M-PLUS Executive Reference Manual", Digital Equipment Corporation, Maynard, M.A., 1979

[S76] Schmid H. A., "On Efficient Implementation of Conditional Critical Regions and the Construction of Monitors", Acta Informatica, no. 6, 1976, s. 227-249

[W77] Wirth N., "Modula, a Programming Language for Modular Multiprogramming", Software - Practice and Experience, 1977, 7, s. 115-126

3.4. CONCURRENCY

Implemented concurrency mechanisms differ much from those described in the LOGLAN-82 report. In particular, only distributed processes are implemented, so they cannot communicate through shared variables. For this reason semaphores had to be replaced by an entirely new communication mechanism. Such a mechanism has been designed and it is based on the rendez-vous schema.

3.4.1. INVOKING THE LOGLAN INTERPRETER FOR CONCURRENT PROGRAMS

A concurrent LOGLAN program may run on a single computer with concurrency simulated by time slicing. In this case LOGLAN interpreter is invoked as usual. One must only remember that /m optional parameter (see 1.4.) denotes memory size for each process rather than for the whole program.

To achieve true parallel (multiprocessor) execution, a network of IBM PC computers may be used. For the time being, only D-Link Network Version 3.21 is supported. In order to run a LOGLAN program in the network environment take the following steps:

1) make sure that every node is logged on,
2) select arbitrarily one node as a console,
3) invoke the LOGLAN interpreter on every node except the console, giving it /r option with the console node number (see 1.4.). You must give the same program file to all interpreters. Most conveniently it may be achieved by accessing a file on a disk connected through the network to each node.
4) invoke the interpreter on the console without the /r option (in the usual way). Give it the same program file as above.

After the last step the main program process begins its execution on the console node. Other processes may be created dynamically on any node on which an interpreter is running.

Regardless of the fact whether the network is used or not, more than one process may be executed on the same computer.

3.4.2. RESTRICTIONS AND DIFFERENCES FROM THE REPORT

All processes (even those executed on the same computer) are implemented as distributed, i.e. without any shared memory. This fact implies some restrictions on how processes may be used. Not all restrictions are enforced by the present compiler, so it is the programmer's responsibility to respect them. This is the list of restrictions:

1) all process units must be declared as global, i.e. directly inside the main program block
2) a process cannot access global variables (except for the main program process)
3) any remote access to a process object other than a procedure (or function) call is inhibited
4) each parameter of
   - a process
   - a procedure called by remote access to a process object
   - a procedure parameter of a process

   must be one of the following:
   - a value of the primitive type (INTEGER, REAL, CHAR, BOOLEAN, STRING)
   - a procedure declared directly inside a process
   - a procedure which is a formal parameter of a process
   - any reference to a process object.

   This restriction implies that references to objects other than processes have only local meaning (in a single process) and cannot be passed among the processes.
5) comparisons, IS, IN and QUA operations are not allowed for the references to processes.
6) operations which require dynamic type checking on the

references to processes are not allowed

7) a process may be attached only by a proper coroutine (not by a process) generated by it

8) the variable MAIN is accesible only in the main program process.

The following concurrent constructs described in the report are not implemented at all:

- semaphores and all operations on them
- the WAIT expression.

Semantics of the NEW generator is slightly modified when applied to the processes. The first parameter of the first process unit in the prefix sequence must be of type INTEGER. This parameter denotes the node number of the computer on which this process will be created. For a single computer operation this parameter must be equal to 0.

Example:

```
unit A:class(msg:string);
...
end A;
unit P:A process(node:integer, pi:real);
...
end P;
...
var x:P;
...
begin
...
 (* Create process on node 4. The first parameter is the *)
 (* string required by the prefix A, the second is the node *)
 (* number *)
 x := new P("Hello", 4, 3.141592653);
...
end
```

The following parallel constructs are implemented as defined in

the report:

- KILL operation for a process
- RESUME statement
- STOP statement without parameter.

3.4.3. COMMUNICATION MECHANISM

Processes may communicate and synchronize by a mechanism based on rendez-vous. It will be referred to as "alien call" in the following description.

An alien call is either:
- a procedure (or function) call performed by a remote access to a process object, or
- a call of a procedure which is a formal parameter of a process, or
- a call of a procedure which is a formal parameter of an alien-called procedure (this is a recursive definition).

Every process object has an enable mask. It is defined as a subset of all procedures declared directly inside a process unit or any unit from its prefix sequence (i.e. subset of all procedures that may be alien-called).

A procedure is enabled in a process if it belongs to that process' enable mask. A procedure is disabled if it does not belong to the enable mask.

Immediately after generation of a process object its enable mask is empty (all procedures are disabled).

Semantics of the alien call is different from the remote call described in the report. Both the calling process and the process in which the procedure is declared (i.e. the called process) are involved in the alien call. This way the alien call may be used as a synchronization mechanism.

The calling process passes the input parameters and waits for the call to be completed.

The alien-called procedure is executed by the called process. Execution of the procedure will not begin before certain

conditions are satisfied. First, the called process must not be suspended in any way. The only exception is that it may be waiting during the ACCEPT statement (see below). Second, the procedure must be enabled in the called process.

When the above two conditions are met the called process is interrupted and forced to execute the alien-called procedure (with parameters passed by the calling process).

Upon entry to the alien-called procedure all procedures become disabled in the called process.

Upon exit the enable mask of the called process is restored to that from before the call (regardless of how it has been changed during the execution of the procedure). The called process is resumed at the point of the interruption. The execution of the ACCEPT statement is ended if the called process was waiting during the ACCEPT (see below). At last the calling process reads back the output parameters and resumes its execution after the call statement.

The process executing an alien-called procedure can easily be interrupted by another alien call if the enable mask is changed.

There are some new language constructs associated with the alien call mechanism. The following statements change the enable mask of a process:

```
ENABLE p1, ..., pn
```

enables the procedures with identifiers p1, ..., pn. If there are any processes waiting for an alien call of one of these procedures, one of them is chosen and its request is processed. The scheduling is done on a FIFO basis, so it is strongly fair. The statement:

```
DISABLE p1, ..., pn
```

disables the procedures with identifiers p1, ..., pn.

In addition a special form of the RETURN statement:

```
RETURN ENABLE p1, ..., pn DISABLE q1, ..., qn
```

allows to enable the procedures p1, ..., pn and disable the procedures q1,...,qn after the enable mask is restored on exit from the alien-called procedure. It is legal only in the alien-called procedures (the legality is not enforced by the compiler).

A called process may avoid busy waiting for an alien call by means of the ACCEPT statement:

```
ACCEPT p1, ..., pn
```

adds the procedures p1, ..., pn to the current mask, and waits for an alien call of one of the currently enabled procedures. After the procedure return the enable mask is restored to that from before the ACCEPT statement.

Note that the ACCEPT statement alone (i.e. without any ENABLE/DISABLE statements or options) provides a sufficient communication mechanism. In this case the called process may execute the alien-called procedure only during the ACCEPT statement (because otherwise all procedures are disabled). It means that the enable mask may be forgotten altogether and the alien call may be used as a pure totally synchronous rendez-vous. Other constructs are introduced to make partially asynchronous communication patterns possible.