Zbigniew Weiss
Tadeusz Gruźlewski

PROGRAMOWANIE WSPÓŁBIEŻNE I ROZPROSZONE

w przykładach i zadaniach

WYDAWNICTWA NAUKOWO-TECHNICZNE

Spis treści

1. Wstęp

Programowanie - kiedyś wielka umiejętność - dziś staje się zwykłym rzemiosłem. Co więcej, są już systemy umożliwiające automatyczne tworzenie kompilatorów czy oprogramowania baz danych bezpośrednio z ich specyfikacji. Licznie pojawiające się systemy wspomagające typu CASE (Computer Aided Software Engineering) umożliwiają częściowe zautomatyzowanie etapu programowania także w innych dziedzinach. Wygląda na to, że umiejętność programowania w języku wysokiego poziomu będzie w przyszłości potrzebna niewielkiej grupie specjalistów (tak jak dziś umiejętność programowania w języku asemblera).

Jest jednak pewna dziedzina programowania, która dotychczas nie poddaje się łatwo automatyzacji. To programowanie współbieżne. Metody specyfikacji, weryfikacji i dowodzenia poprawności są dla tego typu programowania znacznie bardziej skomplikowane niż dla programowania sekwencyjnego. Współbieżność wymaga uwzględnienia trudnych do opisania zależności czasowych między programami. Brakuje przy tym metod testowania, które pozwoliłyby na szybkie wykrycie błędów w synchronizacji. Zachowanie się programu współbieżnego może bowiem zależeć od takich czynników zewnętrznych, jak prędkość procesora czy sposób zarządzania nim.

Z drugiej strony należy podkreślić, że programowanie współbieżne stosuje się dziś powszechnie, poczynając od takich klasycznych dziedzin informatyki jak systemy operacyjne (na których gruncie powstało), przez obliczenia równoległe, systemy zarządzania baz danych, wbudowane systemy czasu rzeczywistego (sterowanie procesami technologicznymi, monitorowanie pacjentów w szpitalu, sterowanie pracą elektrowni atomowej lub lotem samolotu), a na grach komputerowych kończąc. Ma ono również duże znaczenie w dyscyplinach ściśle związanych z informatyką, takich jak na przykład telekomunikacja. Można zatem zaryzykować stwierdzenie, że programowanie współbieżne będzie jeszcze przez długi czas cenioną umiejętnością.

Początkowo programowanie współbieżne powstało na potrzeby systemów scentralizowanych (tzn. ze wspólną pamięcią), jednak obecnie nabiera jeszcze większego znaczenia, ze względu na powszechne stosowanie sieci komputerowych i systemów rozproszonych.

Czasami dla podkreślenia faktu, że chodzi o programowanie współbieżne dla tego rodzaju systemów, nazywa się je programowaniem rozproszonym.

W istocie współbieżność nie jest czymś niezwykłym, przeciwnie, to sekwencyjne programowanie jest ograniczeniem wynikającym z naszego sekwencyjnego myślenia o problemach. Przecież współbieżność i rozproszoność charakteryzują również naszą własną działalność. Każdy z nas codziennie załatwia współbieżnie wiele własnych spraw, są też sprawy załatwiane współbieżnie przez wiele osób. Programy komputerowe, które mają być również naszymi partnerami i pomocnikami w codziennym życiu, powinny odzwierciedlać takie zachowanie, muszą więc być z natury współbieżne i rozproszone.

Świadczą o tym kierunki rozwoju informatyki zarówno w Polsce, jak i na świecie. Lata osiemdziesiąte były okresem dominowania komputerów osobistych i stacji roboczych. Obecnie do profesjonalnych obliczeń coraz częściej stosuje się komputery wieloprogramowe połączone w sieci komputerowe. W różnych ośrodkach akademickich powstają eksperymentalne rozproszone systemy operacyjne zarządzające sieciami komputerowymi tak, jakby był to jeden wielki komputer. Wraz z tymi zmianami zwiększają się możliwości i dziedziny zastosowań programowania współbieżnego i rozproszonego.

Łączenie ze sobą w sieci komputerów o różnej architekturze spowodowało ogromny wzrost popularności systemu operacyjnego Unix. W systemie tym istnieją bardzo rozbudowane mechanizmy do programowania współbieżnego i rozproszonego. W książce tej poświęcamy im sporo miejsca.

Programowanie współbieżne znajduje się w programach studiów informatycznych na całym świecie albo jako odrębny przedmiot, albo jako bardzo rozbudowana część wykładu z systemów operacyjnych. Również w Polsce od wielu lat programowaniu współbieżnemu w nauczaniu informatyki poświęca się wiele uwagi. Na przykład w obowiązującym do niedawna programie uniwersyteckich studiów informatycznych w Warszawie na programowanie współbieżne przeznaczano około połowy czasu przedmiotu „Systemy operacyjne" (czyli 30 godzin wykładu i 30 godzin ćwiczeń). W nowym programie studiów, opracowanym w Instytucie Informatyki Uniwersytetu Warszawskiego, jest to już całkowicie odrębny przedmiot w wymiarze 90 godzin (po 30 godzin wykładu, ćwiczeń i pracowni). Świadczy to dobitnie o tym, jak dużą wagę przywiązuje się obecnie do zagadnień programowania współbieżnego.

Zasadniczą część niniejszej książki stanowi zbiór zadań z rozwiązaniami. Zadania dotyczą szerokiego wachlarza problemów i zastosowań, takich jak Unix jest znakiem zastrzeżonym Bell Laboratories. zarządzanie zasobami w systemach operacyjnych (np. priorytetowy dostęp do zasobów, szeregowanie żądań do dysku, synchroniczne i asynchroniczne żądania do dysku, zarządzanie pamięcią operacyjną, unikanie blokady), synchronizacja procesów w systemach scentralizowanych i rozproszonych (np. elekcja, korygowanie logicznych zegarów, komunikacja przez sieć), kontrola współbieżności transakcji w bazach danych, algorytmy współbieżne (np. współbieżne sortowanie, wyszukiwanie wzorca, wyznaczanie wartości wielomianu, mnożenie wielomianów i macierzy, wyznaczanie histogramu).

Zamieszczone w książce zadania były wykorzystywane przez autorów na prowadzonych przez nich w ciągu ostatnich kilku lat wykładach i ćwiczeniach z programowania współbieżnego dla studentów IV-go roku informatyki Uniwersytetu Warszawskiego. Większość tych zadań wymyślono specjalnie na te zajęcia. Pozostałe to klasyczne zadania publikowane już w wielu podręcznikach poświęconych współbieżności, przytaczamy je tu jednak, gdyż są istotnym uzupełnieniem książki.

W książce podajemy podstawowe pojęcia i problemy związane z programowaniem współbieżnym i rozproszonym oraz obszernie omawiamy podstawowe mechanizmy do synchronizacji i komunikacji, takie jak semafory, monitory, spotkania i przestrzeń krotek. Szczególną uwagę zwracamy na mechanizmy programowania współbieżnego dostępne w systemie Unix, takie jak semafory, komunikaty i kanały oraz zdalne wywalanie procedury, które dotychczas nie są dostatecznie dobrze wyjaśnione w literaturze dostępnej w języku polskim.

Rozdział 2 zawiera definicje pojęć potrzebnych do zrozumienia przykładów i rozwiązania zadań zamieszczonych w książce. Na przykładach wziętych z życia wyjaśniamy abstrakcyjne pojęcia procesu, współbieżności, sekcji krytycznej, obiektu dzielonego, synchronizacji. Definiujemy cztery klasyczne problemy współbieżności: wzajemnego wykluczania, producenta i konsumenta, czytelników i pisarzy oraz pięciu filozofów. Będziemy się do nich odwoływać ilustrując użycie mechanizmów programowania współbieżnego prezentowanych w dalszej części książki. Zwracamy uwagę na zjawiska blokady i zagłodzenia oraz wiążące się z tym pojęcia bezpieczeństwa i żywotności programów współbieżnych. Omawiamy niskopoziomowe mechanizmy służące do realizacji wzajemnego wykluczania: przerwania, sprzętowo gwarantowane wzajemne wykluczanie przy dostępie do komórki pamięci (i wiążący się z tym algorytm Dekkera) oraz instrukcje typu Test&Set. Podając wiele przykładów z życia pragniemy

wyrobić u czytelnika pewne intuicje co do wysokopoziomowych mechanizmów synchronizacji gwarantowanych na poziomie języka programowania. Okazuje się bowiem, że mechanizmy te bardzo przypominają sposoby, jakimi my sami ze sobą się synchronizujemy i komunikujemy.

Wszystkie rozdziały począwszy od trzeciego mają jednakową strukturę. Zaczyna je szczegółowy opis wybranego mechanizmu. Następnie pokazujemy sposób jego użycia na przykładzie różnych wariantów rozwiązań klasycznych problemów omówionych w rozdz. 2. Trzecią częścią każdego rozdziału są treści zadań do samodzielnego rozwiązania. W czwartej zaś podajemy przykładowe ich rozwiązania wraz z niezbędnymi wyjaśnieniami.

Rozdział 3 jest poświęcony synchronizacji procesów w systemach scentralizowanych za pomocą klasycznego semafora Dijkstry.

W **rozdziale 4** zajmujemy się synchronizacją procesów za pomocą monitorów. Prezentujemy w nim także sposoby zapisywania monitora w różnych językach programowania.

W **rozdziale 5** omawiamy język programowania współbieżnego CSP oparty na mechanizmie symetrycznych spotkań zaproponowanym przez C. A. R. Hoare'a oraz informujemy o trzech językach wyrosłych z CSP, w których także używa się mechanizmu symetrycznych spotkań. Są to: ~~occam (przeznaczony do programowania transputerów), Paralld C (rozszerzenie języka C o mechanizm spotkań) oraz Edip (do programowania sieci mikrokomputerowych, zaprojektowany w Instytucie Informatyki UW i zaimplementowany na komputerach typu IBM PC pod systemem MS-DOS).

Rozdział 6 dotyczy asymetrycznych spotkań w języku Ada. Ograniczyliśmy się do krótkiego omówienia tylko tych konstrukcji języka, których użyto w przykładach i rozwiązaniach.

W **rozdziale 7** zajmujemy się przestrzenią krotek zaproponowaną w eksperymentalnym języku Linda. Mechanizm ten pozwala na asynchroniczną komunikację między procesami, ale jego główną zaletą i siłą jest możliwość selektywnego wyboru komunikatów.

W rozdziałach 3-7 omawiamy mechanizmy synchronizacji i komunikacji stosowane w różnych językach programowania współbieżnego niezależnie od systemu operacyjnego. W rozdziałach 8-10 zajmujemy się mechanizmami dostępnymi na poziomie konkretnego systemu operacyjnego, którym w tym przypadku jest Unix. Wybór tego systemu był podyktowany jego dużą popularnością, przenośnością, otwartością a także faktem, że zawiera on bardzo bogaty zestaw narzędzi wspomagających programowanie współbieżne.

W **rozdziale 8** prezentujemy mechanizm semaforów dostępny w systemie Unix. Semafory unixowe różnią się istotnie od klasycznych semaforów Dijkstry. Są znacznie silniejsze, zawierają w sobie różne uogólnienia klasycznej definicji. Aby zyskać na zwięzłości i przejrzystości, w prezentowanych przykładach i w rozwiązaniach zadań posługujemy się specjalnie wprowadzoną notacją. Kilka przykładów zapisujemy całkowicie w języku C, by pokazać, jak naprawdę powinien wyglądać program współbieżny w systemie Unix. (W podobny sposób postępujemy w rozdz. 9 i 10.)

Rozdział 9 jest poświęcony mechanizmowi komunikatów i kanałów służącemu w systemie Unix do asynchronicznego komunikowania się procesów na jednej maszynie. Mechanizm ten jest uogólnieniem mechanizmu potoków, dlatego o samych potokach w systemie Unix tylko wspominamy.

W **rozdziale 10** przedstawiamy mechanizm zdalnego wywołania procedury, znany też jako RPC (Remote Procedure Call), który jest uogólnieniem zwykłego wywołania procedury w przypadku systemów rozproszonych. W systemie Unix jest to podstawowy mechanizm wysokiego poziomu przeznaczony głównie do komunikacji między procesami wykonywanymi na fizycznie różnych maszynach. Omówione wcześniej semafory i komunikaty służą do synchronizacji procesów wykonujących się na jednej maszynie.

W ostatnim rozdziale pokazujemy, jak można ćwiczyć programowanie współbieżne nie mając do dyspozycji żadnego języka programowania współbieżnego i pracując pod systemem operacyjnym MS-DOS, który także nie zapewnia żadnej współbieżności. Omawiamy tu mechanizm potoków w systemie MS-DOS podkreślając jednocześnie istotną różnicę między tym mechanizmem a potokami w Unixie.

Niektóre zadania omówione we wcześniejszych rozdziałach proponujemy także do rozwiązania w rozdziałach późniejszych, gdyż ze względu na użyty mechanizm ich rozwiązania mogą się znacznie różnić. Aby ułatwić czytelnikowi zorientowanie się, za pomocą jakich mechanizmów rozwiązano poszczególne problemy i zadania, zamieszczamy na końcu ich alfabetyczny spis.

Wykaz cytowanej literatury poprzedzamy krótkim omówieniem.

Podziękowania

Pragniemy podziękować naszym współpracownikom, z którymi prowadziliśmy przez wiele lat zajęcia z systemów operacyjnych na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego: prof. Janowi Madeyowi, dr Janinie Mincer-Daszkiewicz i dr-owi Kazimierzowi Grygielowi. Pomogli oni nam w ostatecznym sformułowaniu wielu zamieszczonych tu zadań. Szczególnie gorąco dziękujemy prof. Janowi Madeyowi i dr Janinie Mincer-Daszkiewicz, którzy byli również pierwszymi recenzentami tej książki. Dziękujemy także wszystkim studentom Instytutu Informatyki (zwłaszcza panu Krzysztofowi Stenclowi), którzy mieli z nami zajęcia, za to, że starali się, jak mogli, by te zadania rozwiązać. Podsunęli nam wiele interesujących pomysłów, które wykorzystaliśmy w proponowanych tu rozwiązaniach. Pokazali także, gdzie leżą źródła najczęstszych błędów, dzięki czemu możemy ostrzec przed nimi naszych czytelników. Gorące podziękowania należą się firmie PQ Information Group z Holandii, na której sprzęcie były testowane wszystkie zawarte w tej książce przykłady programów w języku C pod systemem Unix.

Warszawa 1992

2. Podstawowe pojęcia i problemy

2.1 Proces

Omawiane w tym rozdziale pojęcia i problemy można znaleźć prawie w każdej książce poświęconej programowaniu współbieżnemu i w większości książek poświęconych systemom operacyjnym. Dlatego ograniczymy się tu do przytoczenia podstawowych definicji. Jednak, aby nie była to li tylko sucha prezentacja faktów, będziemy starali się podać także pewne intuicje odwołując się do przykładów wziętych z życia codziennego. Słowa proces używamy w wielu dziedzinach naszego życia. Mówimy np. o procesie sądowym, procesie produkcyjnym, procesie dojrzewania, procesie rozpadu itd. W każdej sytuacji mamy na myśli pewną sekwencję zmian dokonujących się zgodnie z określonym schematem. Sekwencja ta może być skończona, jak np. w procesie produkcyjnym, lub (potencjalnie) nieskończona, jak np. w procesie rozwoju życia na Ziemi.

W informatyce pojęcie to pojawiło się po raz pierwszy w związki! z systemami operacyjnymi. Przez proces rozumie się program (sekwencyjny) w trakcie wykonywania1. Ponieważ program jest sformalizowanym zapisem algorytmu, więc proces jest sekwencją zmian stanu systemu komputerowego, które odbywają się zgodnie z tym algorytmem. Odpowiednich zmian dokonuje urządzenie zwane procesorem, natomiast kod programu jest przechowywany w pamięci operacyjnej. Procesor i pamięć operacyjna to dwa urządzenia niezbędne do wykonywania każdego procesu.

Co do definicji procesu nie ma wśród informatyków powszechnej zgody. Przyjęta przez nas definicja jest bardzo ogólna i zawiera w sobie bardziej szczegółowe definicje innych autorów.

Procesy mogą być skończone i nieskończone. Każdy początkujący programista zapewne przynajmniej raz uruchomił proces nieskończony wtedy, gdy spróbował wykonać program zawierający nieskończoną pętlę. Pętla, taka powstaje zazwyczaj wskutek błędnego sformułowania warunku. Trzeba wówczas sięgać po środki ostateczne, jak wyłączenia komputera lub (w przypadku większych systemów) zwrócenie się o pomoc do operatora. Ale procesy nieskończone nie zawsze wiążą się z błędnym programowaniem. Pewne procesy z istoty rzeczy muszą być nieskończone. Przykładem jest system operacyjny. Czytelnik byłby zapewne bardzo zdziwiony, gdyby system operacyjny jego ulubionego komputera osobistego poinformował go, że właśnie się zakończył i nie przyjmie już żadnych nowych poleceń. Takie zachowanie uznałby za poważny błąd. Od systemu operacyjnego wymagamy, bowiem, aby trwał nieskończenie, gotowy na każde nasze żądanie. Podobnie procesy w systemach wbudowanych, zwanych inaczej systemami czasu rzeczywistego (np. procesy sterujące różnego rodzaju automatami produkcyjnymi), muszą także być z natury rzeczy nieskończone.
Tak naprawdę nie ma większych różnic między pojęciem procesu w systemie komputerowym a pojęciem procesu w innych dziedzinach. W zależności od tego, według jakiego algorytmu jest wykonywany proces komputerowy, możemy mówić np. o procesie przetwarzania danych, procesie obliczeniowym czy procesie elektronicznego składania tekstu.

2.2 Procesy współbieżne

Mówimy, że dwa procesy są współbieżne, jeśli jeden z nich rozpoczyna się przed zakończeniem drugiego.

Zauważmy, że zgodnie z tą definicją proces nieskończony jest współbieżny ze wszystkimi procesami, które rozpoczęły się od niego później. W szczególności system operacyjny, a dokładniej procesy wchodzące w jego skład wykonują się współbieżnie ze wszystkimi procesami uruchamianymi przez użytkowników systemu komputerowego.

2.2.1 Podział czasu

Napisaliśmy, że realizacją procesu zajmuje się procesor. W wielu systemach komputerowych jest tylko jeden procesor. Mimo to można w nich realizować współbieżne wykonanie wielu procesów. Uzyskuje się to dzięki zasadzie podziału czasu. Zasadę tę stosuje w życiu każdy z nas. Dobrym przykładem jest uczenie się różnych przedmiotów. Nie uczymy się przecież w szkole najpierw samej matematyki, potem samej historii itd. Procesy nauki przeplatają się, a rozkład zajęć określa, kiedy i ile czasu uczeń poświęci na naukę danego przedmiotu.

W jednoprocesorowych systemach komputerowych czas pracy procesora jest dzielony między wszystkie wykonywane współbieżnie procesy. Problem, ile czasu przydzielić, komu i kiedy, jest przedmiotem rozważań w teorii systemów operacyjnych (patrz np. [S1PG91]).

2.2.2 Jednoczesność

W jednoprocesorowych systemach z podziałem czasu w danej chwili może być wykonywany tylko jeden proces, choć rozpoczętych może ich być wiele. Jeśli w systemie komputerowym jest wiele procesorów, mogą one wykonywać różne procesy jednocześnie (równolegle), a zatem w takim systemie w danej chwili może być wykonywanych kilka procesów.

W języku potocznym, gdy mówimy „Oni jednocześnie skoczyli do wody", mamy na myśli, że oba skoki zdarzyły się w tej samej chwili. Natomiast, gdy mówimy „On zajmuje się jednocześnie dwoma sprawami", mamy na myśli, że on dzieli swój czas między obie te sprawy. W używanej tu terminologii słowo jednoczesny ma znaczenie ograniczone tylko do tego pierwszego przypadku. Do określenia obu podanych sytuacji będziemy używać słowa współbieżny.

2.2.3 Komunikacja i synchronizacja

Według podanej definicji, współbieżne są np. procesy wykonywane jednocześnie na dwóch nie połączonych ze sobą komputerach osobistych. Ze stwierdzenia tego faktu nic jednak dalej nie wynika, procesy te, bowiem nie mają na siebie żadnego wpływu.

Dalej będą nas interesować jedynie takie procesy, których wykonania są od siebie uzależnione. Uzależnienie takie może wynikać z faktu, że procesy ze sobą współpracują lub między sobą współzawodniczą. Współpraca wymaga od procesów komunikowania się — do działania jednego z nich jest potrzebna informacja tworzona przez drugi. Akcje komunikujących się procesów muszą być częściowo uporządkowane w czasie, ponieważ informacja musi najpierw zostać utworzona, zanim zostanie wykorzystana. Takie porządkowanie w czasie akcji różnych procesów nazywa się synchronizowaniem. Współzawodnictwo także wymaga synchronizacji — akcja procesu musi być wstrzymana, jeśli zasób potrzebny do jej wykonania jest w danej chwili zajęty przez inny proces.

2.2.4 Program współbieżny

Można wyróżnić dwa poziomy wzajemnego uzależnienia procesów współbieżnych. Pierwszy stanowią procesy będące współbieżnym wykonaniem niezależnych programów użytkowych. Procesy takie nie współpracują, a jedynie współzawodniczą między sobą wymagając synchronizacji przy dostępie do różnych urządzeń komputera. Synchronizacją tą zajmuje się system operacyjny, programista nie ma na nią zazwyczaj bezpośredniego wpływu.

Poziom drugi stanowią procesy utworzone w obrębie jednego programu. Procesy te komunikują się i synchronizują w sposób określony przez programistę. Program opisujący zachowanie się zbioru takich współbieżnych procesów nazywa się programem współbieżnym. Zachowanie się pojedynczego procesu opisuje w programie współbieżnym jednostka syntaktyczna zwana także procesem2. Program współbieżny jest, więc zbiorem procesów — jednostek syntaktycznych. Natomiast program współbieżny w trakcie wykonania jest zbiorem procesów, z których każdy jest wykonaniem procesu — jednostki syntaktycznej. Ponieważ mamy tu do czynienia z odwzorowaniem wzajemnie jednoznacznym, będziemy dalej używać słowa proces w obu znaczeniach.

Typowym przykładem programu współbieżnego jest system operacyjny, który w rzeczywistości składa się z wielu współpracujących ze sobą procesów obsługujących różne zasoby systemu komputerowego.

Przedmiotem niniejszej książki jest programowanie współbieżne, przez co rozumiemy tu nie tylko samo kodowanie w języku programowania współbieżnego, ale także projektowanie sposobu komunikacji i synchronizacji między procesami. Wśród podawanych tu przykładów i zadań znajdą się cztery rodzaje problemów. Po pierwsze, będą to abstrakcje rzeczywistych problemów synchronizacyjnych, a więc różne warianty problemów producenta i konsumenta, czytelników i pisarzy oraz pięciu filozofów. Po drugie, będziemy zajmować się zagadnieniami związanymi z projektowaniem systemów operacyjnych (np. 4.3.12, 4.3.15, 4.3.16, 4.3.17) i systemów baz danych (np. 4.3.5, 5.3.2, 10.3.6). Po trzecie, pokażemy jak można stosować programowanie współbieżne do szybszej, a niejednokrotnie prostszej realizacji pewnych typowych obliczeń, jak np. sortowanie (por. 5.3.15, 5.3.16), wyznaczanie współczynników dwumianu Newtona (por. 5.3.10, 11.3.2), mnożenie wielomianów i macierzy (por. 5.3.11, 5.3.13, 7.3.7), wyznaczanie liczb pierwszych por. 5.3.14, 6.3.8), całkowanie (por. 7.3.9). Po czwarte, będziemy zajmowali się problemami wziętymi z życia codziennego, które nie zawsze mają swe odpowiedniki w teorii systemów operacyjnych, a których rozwiązanie może okazać się ciekawym ćwiczeniem (np. 3.3.9, 4.3.6, 4.3.9, 7.3.8).

2.2.5 Wątki

Oprócz pojęcia procesu ostatnio wprowadza się też pojęcie wątku (thread), por. [Gosc91, Nels91, S1PG91, Tane92]. Różnica między procesem a wątkiem polega przede wszystkim na sposobie wykorzystywania dostępnych zasobów. Każdy proces ma przydzielony odrębny obszar pamięci operacyjnej

Wyjątkiem jest język Ada, w którym mówi się o współbieżnych zadaniach (taskach).i współzawodniczy z innymi procesami o dostęp do zasobów komputera. Często takie wyraźne rozgraniczenie między procesami nie jest potrzebne. Znacznie taniej jest wówczas wykonywać je we wspólnej przestrzeni adresowej tworząc współbieżnie działające wątki. Wątek jest podstawową jednostką wykorzystującą procesor. Grupa równorzędnych wątków współdzieli przestrzeń adresową, kod i zasoby systemu operacyjnego.

Środowisko, w którym wykonują się wątki, nazywa się zadaniem (task). W tym rozumieniu proces jest zadaniem, w którym wykonuje się tylko jeden wątek. Wątki pozwalają na współbieżną realizację wielu żądań skierowanych do tego samego zadania. Wątki mogą być synchroniczne, gdy same sobie przekazują sterowanie, lub asynchroniczne. W tym drugim przypadku jest potrzebny mechanizm umożliwiający synchronizację przy dostępie do współdzielonych danych. Omawiane w tej książce mechanizmy synchronizacji mogą służyć zarówno do synchronizacji procesów, jak i wątków. W dalszym ciągu będziemy, więc mówić jedynie o procesach.

2.3 Wzajemne wykluczanie

2.3.1 Zasób dzielony i sekcja krytyczna

Napisaliśmy wcześniej, że procesy współbieżne mogą ze sobą współzawodniczyć o dostęp do wspólnie użytkowanych zasobów. Chodzi tu o takie zasoby, które w danej chwili mogą być wykorzystywane tylko przez jeden proces (lub ograniczoną ich liczbę, mniejszą od liczby chętnych). Jest to sytuacja dość często spotykana w życiu. Każdemu zdarzyło się, że chciał skorzystać z łazienki właśnie wtedy, gdy była ona zajęta, zadzwonić, gdy ktoś już rozmawiał przez telefon, zrobić zakupy w sklepie samoobsługowym, gdy wszystkie koszyki były w użyciu. Skądinąd wiadomo, że w każdej takiej sytuacji trzeba po prostu zaczekać (na zwolnienie łazienki, zakończenie rozmowy, czy zwrot koszyka). Natomiast wtedy, gdy dwie osoby jednocześnie chcą wejść do pustej łazienki, zadzwonić z tego samego telefonu, czy wziąć ten sam koszyk, trzeba zastosować zasadę uprzejmości i dobrych obyczajów.

W teorii procesów współbieżnych wspólny obiekt, z którego może korzystać w sposób wyłączny wiele procesów (np. łazienka, telefon, koszyk) nazywa się zasobem dzielonym, natomiast fragment procesu, w którym korzysta on z obiektu dzielonego (mycie się, telefonowanie, zakupy), nazywa się sekcją krytyczną tego procesu.

Ponieważ w danej chwili z obiektu dzielonego może korzystać tylko jeden proces, wykonując swoją sekcję krytyczną uniemożliwia on wykonanie sekcji krytycznych innym procesom.

2.3.2 Problem wzajemnego wykluczania

Problem wzajemnego wykluczania definiuje się następująco: zsynchronizować N procesów, z których każdy w nieskończonej pętli na przemian zajmuje się własnymi sprawami i wykonuje sekcję krytyczną, w taki sposób, aby wykonanie sekcji krytycznych jakichkolwiek dwóch lub więcej procesów nie pokrywało się w czasie.

Aby ten problem rozwiązać, należy do treści każdego procesu wprowadzić, dodatkowe instrukcje poprzedzające sekcję krytyczną (nazywa się je protokołem wstępnym) i instrukcje następujące bezpośrednio po sekcji krytycznej (protokół końcowy). Protokół wstępny i końcowy to po prostu programowa realizacja czekania i stosowanej w życiu zasady uprzejmości.

Rozwiązania problemu wzajemnego wykluczania za pomocą mechanizmów sprzętowych pokażemy w dalszej części tego rozdziału. W następnych rozdziałach przedstawimy rozwiązania tego problemu za pomocą różnych mechanizmów wysokiego poziomu.

2.3.3 Wymagania czasowe

Problem wzajemnego wykluczania można rozwiązać w sposób satysfakcjonujący wszystkie procesy, ale w tym celu należy poczynić pewne założenia, co do zachowania się samych procesów.

Po pierwsze, żaden proces nie może wykonywać swej sekcji krytycznej nieskończenie długo, a zwłaszcza nie może on się wewnątrz niej zapętlić lub zakończyć w wyniku jakiegoś błędu (np. nikt nie umrze w łazience czy przy telefonie ani nie wyniesie koszyka ze sklepu). Uniemożliwiłby on innym procesom wejście3 do sekcji krytycznej. Generalna zasada jest taka., że w sekcji krytycznej proces powinien przebywać możliwie najkrócej.

Taki sam warunek dotyczy protokołów wstępnego i końcowego, które w praktyce należy traktować jako część sekcji krytycznej. Zakładamy, więc, że w skończonym czasie proces wykona protokół wstępny i (gdy uzyska pozwolenie) wejdzie do sekcji krytycznej, po czym ją opuści i wykona protokół końcowy.

Po drugie, zachowanie się procesów poza sekcją krytyczną nie powinno być w żaden sposób ograniczone. Poza sekcją krytyczną proces może się, więc zapętlić lub skończyć.

Po trzecie, procesy mogą się wykonywać z różnymi dowolnie wybranymi prędkościami. Przez wejście do sekcji krytycznej rozumie się rozpoczęcie jej wykonywania.

2.4 Bezpieczeństwo i żywotność

2.4.1 Własność bezpieczeństwa

W celu wykazania, że program sekwencyjny jest poprawny, należy udowodnić dwie rzeczy. Po pierwsze, zawsze, jeśli program się zatrzyma, to zwróci nam dobre wyniki (tę własność programu nazywa się częściową poprawnością); po drugie, że w ogóle się zatrzyma (nazywa się to własnością stopu). Programom współbieżnym nie stawia się jednak takich samych wymagań — mogą one przecież w ogóle się nie zatrzymywać.

Własność bezpieczeństwa jest uogólnieniem własności częściowej poprawności na programy współbieżne. Program współbieżny jest bezpieczny, jeśli nigdy nie doprowadza do niepożądanego stanu lub, inaczej mówiąc, zawsze utrzymuje system w pożądanym stanie. W przypadku problemu wzajemnego wykluczania własność bezpieczeństwa oznacza, że nigdy dwa procesy nie znajdą się jednocześnie w swoich sekcjach krytycznych.

Dowodzenie częściowej poprawności nawet bardzo prostych programów sekwencyjnych nie jest rzeczą łatwą, dowodzenie własności bezpieczeństwa jest jeszcze trudniejsze. W podręcznikach programowania rzadko, kiedy znajdziemy formalne dowody poprawności przytaczanych tam programów. W niniejszej książce także nie dowodzimy własności bezpieczeństwa podawanych rozwiązań. Czytelnika zainteresowanego metodami dowodzenia tej własności odsyłamy do książek [Hare92, BenADO], w których można znaleźć szkice dowodów własności bezpieczeństwa dla rozwiązań kilku klasycznych problemów programowania współbieżnego. Pokazuje się w nich, że żaden nieskończony ciąg akcji procesów przy dowolnym scenariuszu synchronizacji i komunikacji między nimi nie prowadzi do sytuacji niepożądanej. Warto zauważyć, że aby wykazać, iż program współbieżny nie jest bezpieczny, wystarczy wskazać ciąg akcji poszczególnych procesów, które doprowadzają do stanu niepożądanego. W kilku przypadkach celowo przytaczamy w tej książce błędne, choć narzucające się w pierwszej chwili rozwiązania, wskazując jednocześnie, w jaki sposób doprowadzają one do niepożądanej sytuacji (por. 3.2.4, 3.4.1, 5.2.1, 5.2.4, 8.4.1).

2.4.2 Własność żywotności

Własność żywotności jest uogólnieniem własności stopu na programy współbieżne. Program współbieżny jest żywotny, jeśli zapewnia, że każde pożądane zdarzenie w końcu zajdzie. W przypadku problemu wzajemnego wykluczania własność żywotności oznacza, że jeśli jakiś proces czeka na wejście do swojej sekcji krytycznej, to w końcu do niej wejdzie.

Dowodzenie własności żywotności jest łatwiejsze niż dowodzenie własności bezpieczeństwa. Wystarczy pokazać skończony ciąg akcji procesów, które przy każdym scenariuszu synchronizacji między nimi doprowadzą w końcu do pożądanej sytuacji. Natomiast, aby pokazać, że program nie ma własności żywotności, trzeba podać jeden nieskończony ciąg akcji procesów i jeden, być może bardzo szczególnie dobrany, scenariusz synchronizacji, przy którym nigdy nie daje się osiągnąć pożądanej sytuacji. W niniejszej książce będziemy czasami omawiać rozwiązania nie mające własności żywotności, wskazując odpowiedni nieskończony ciąg akcji (por. 4.2.3, 4.2.4, 7.2.3).

2.4.3 Sprawiedliwość

Własność żywotności gwarantuje nam, że zdarzenie, na które czekamy, w końcu zajdzie, ale nie określa, kiedy to się stanie. Na przykład rozwiązanie problemu wzajemnego wykluczania dwóch identycznie zachowujących się procesów, które pozwala jednemu z nich wchodzić do swej sekcji krytycznej, co sekundę, a drugiemu co pięć godzin, ma niewątpliwie własność żywotności i formalnie jest poprawne, ale intuicyjnie trudno nam się z tym pogodzić. Od rozwiązania takiego chcielibyśmy bowiem, aby zapewniało jednakowe traktowanie obu procesów — chcemy by było ono sprawiedliwe (uczciwe).

Gdy procesy nie są identyczne, sprawiedliwość jest cechą, którą trudno zdefiniować, a jeszcze trudniej zmierzyć. Jaka jest sprawiedliwa kolejność wpuszczania procesów do sekcji krytycznej? Czy według kolejności zgłoszeń? A może według planowanego czasu przebywania w sekcji krytycznej? (Te procesy, które chcą być w niej dłużej, powinny dłużej czekać.) A może należy uwzględnić jeszcze, ile czasu procesy czekały podczas poprzednich prób wejścia do sekcji krytycznej?

Możliwości jest wiele i trudno powiedzieć, które podejście jest najbardziej sprawiedliwe. Mimo to w niektórych rozwiązaniach proponowanych w tej książce będziemy starali się uwzględniać i ten aspekt programowania współbieżnego (np. 3.3.4, 3.3.5).

2.5 Blokada i zagłodzenie

2.5.1 Blokada

Jak już pisaliśmy, będziemy zajmować się tylko takimi procesami współbieżnymi, które są ze sobą powiązane przez to, że wymagają wzajemnej synchronizacji lub komunikacji. Oznacza to, że w pewnych sytuacjach procesy będą wstrzymywane w oczekiwaniu na sygnał bądź komunikat od innego procesu. W poprawnym programie współbieżnym w każdym procesie powinno w końcu nastąpić oczekiwane zdarzenie. W niepoprawnym programie mogą wystąpić zjawiska blokady lub zagłodzenia.

Powiemy, że zbiór procesów znajduje się w stanie blokady, jeśli każdy z tych procesów jest wstrzymany w oczekiwaniu na zdarzenie, które może być spowodowane tylko przez jakiś inny proces z tego zbioru.

Zjawisko blokady, zwane także zastojem, zakleszczeniem lub martwym punktem, jest przejawem braku bezpieczeństwa programu, jest to bowiem stan niepożądany. Zjawisko to może wystąpić również w systemie złożonym z procesów, które są powiązane jedynie przez to, że korzystają z tych samych zasobów komputera. W teorii systemów operacyjnych dopracowano się kilku sposobów walki z blokadą. Jednym z nich jest tzw. algorytm bankiera będący przedmiotem zadania 4.3.11. O innych sposobach zapobiegania blokadzie i przełamywania jej można dowiedzieć się m.in. z książek [MaDo83, Shaw79, BiSh89, S1PG91]. O przykładach blokad spoza dziedziny systemów operacyjnych można przeczytać w artykule [WeisOla].

Zauważmy, że jeśli w jakimś programie współbieżnym może wystąpić blokada, nie oznacza to, że wystąpi ona przy każdym wykonaniu tego programu. Dlatego testowanie nie jest dobrą metodą stwierdzania, czy dany zbiór procesów może się zablokować. W podawanych tu zadaniach i rozwiązaniach będziemy zwracali szczególną, uwagę na możliwość wystąpienia blokady i potrzebę odpowiednich przed nią zabezpieczeń (por. 5.3.5, 5.3.6).

Czasami unikanie blokady może być bardzo kosztowne. Jeśli jej wystąpienie jest mało prawdopodobne, lepiej godzić się na nią, ale trzeba wówczas uruchomić mechanizmy jej wykrywanie i usuwania.

2.5.2 Zagłodzenie

Specyficznym przypadkiem nieskończonego wstrzymywania procesu jest zjawisko zagłodzenia zwane także wykluczeniem. Jeśli komunikat lub sygnał synchronizacyjny może być odebrany tylko przez jeden z czekających nań procesów, powstaje problem, który z procesów wybrać. Zjawisko zagłodzenia występuje wówczas, gdy proces nie zostaje wznowiony, mimo że zdarzenie, na które czeka, występuje dowolną liczbę razy. Za każdym razem, gdy proces ten mógłby być wznowiony, jest wybierany jakiś inny czekający proces.

Zagłodzenie jest przejawem braku żywotności programu. Zależy ono od strategii wznawiania procesów. Jeśli procesy będą wznawiane zawsze w kolejności, w jakiej zostały wstrzymane (kolejka prosta), to zjawisko zagłodzenia nie wystąpi. Jeśli jednak o kolejności wznawiania decydują priorytety procesów (kolejka priorytetowa), to jest możliwe, że procesy o niższym priorytecie zostaną zagłodzone przez procesy o wyższym priorytecie.

O ile istnieją algorytmy pozwalające wykrywać zjawisko blokady w trakcie wykonywania programu, o tyle wykrycie zagłodzenia jest praktycznie niemożliwe. Łatwo, co prawda zaobserwować, że pewien proces czeka bardzo długo na jakieś zdarzenie, które wystąpiło już wiele razy, ale nie wiadomo, jak system zachowa się w przyszłości. Można natomiast wykazać, że w programie współbieżnym jest możliwe zagłodzenie, przez pokazanie nieskończonego ciągu zdarzeń w tym programie, w wyniku którego jeden proces (lub więcej) pozostanie na zawsze wstrzymany. (Przykład takiego dowodu znajduje się w książce [Hare92, r.10].)

Mimo że możliwość zagłodzenia świadczy o niepoprawności programu, to często gotowi jesteśmy się z nią pogodzić. (Robimy tak np. wszędzie tam, gdzie stosuje się kolejkę priorytetową.) Wynika to z faktu, że zagłodzenie jest zwykle mało prawdopodobne. (Mało jest np. prawdopodobne, by na miasto wyjechało naraz tyle karetek, aby wstrzymać ruch wszystkich innych pojazdów). Dlatego i w tej książce będziemy przytaczać rozwiązania, które w bardzo szczególnych okolicznościach mogą spowodować zagłodzenie pewnych procesów, a które warto jednak rozważać ze względu na ich prostotę (por. 3.4.2, 4.2.3, 4.2.4).

2.6 Klasyczne problemy współbieżności

2.6.1 Problem producenta i konsumenta

Problem producenta i konsumenta polega na zsynchronizowaniu dwóch procesów: producenta, który cyklicznie produkuje jedną porcję informacji a następnie przekazuje ją do skonsumowania, i konsumenta, który cyklicznie pobiera porcję i konsumuje ją. Jeśli w danej chwili producent nie może pozbyć się wyprodukowanej porcji, musi czekać. Podobnie, jeśli konsument nie może pobrać porcji, także musi czekać. Porcje powinny być konsumowane w kolejności wyprodukowania.

Rozważa się kilka wariantów tego problemu. W najprostszym producent przekazuje porcję bezpośrednio konsumentowi. Najbardziej popularny jest natomiast wariant, w którym zakłada się, że pomiędzy producentem a konsumentem znajduje się An- elementowy bufor (N > 0). Producent wstawia porcję do niepełnego bufora, a konsument pobiera ją z niepustego bufora. W wariancie trzecim (teoretycznym) zakłada się, że bufor jest nieskończony. Procesy producenta i konsumenta muszą tak się synchronizować, aby operacje wstawiania i pobierania z tego samego elementu bufora wzajemnie się wykluczały.

Problem, jak duży powinien być bufor, jest problemem efektywności przyjętego rozwiązania, a jego rozstrzygnięcie zależy od rozkładów czasu produkcji i konsumpcji. Jeśli średni czas konsumpcji jest krótszy od średniego czasu produkcji, bufor może okazać się niepotrzebny. Jeśli jest odwrotnie, dowolnej wielkości bufor okaże się po pewnym czasie za mały. Gdy średnie czasy produkcji i konsumpcji są równe, wielkość bufora dobiera się w zależności od wariancji tych czasów. Warto zaznaczyć, że problem producenta i konsumenta jest abstrakcja wielu rzeczywistych problemów pojawiających się nie tylko w informatyce, ale także w innych dziedzinach takich jak telekomunikacja, transport czy handel.

W niniejszej książce będziemy rozważać także pewne modyfikacje problemu producenta i konsumenta. Uwzględnimy wielu producentów i konsumentów (por. 3.2.2, 4.3.3, 4.3.4), nie będziemy przestrzegać kolejności pobierania porcji (zad. 4.3.4), dopuścimy wstawianie i pobieranie wielu porcji naraz (zad. 4.3.4).

2.6.2 Problem czytelników i pisarzy

Problem czytelników i pisarzy polega na zsynchronizowaniu dwóch grup cyklicznych procesów konkurujących o dostęp do wspólnej czytelni. Proces czytelnik co jakiś czas odczytuje informację zgromadzoną w czytelni i może to robić razem z innymi czytelnikami. Proces pisarz co jakiś czas zapisuje nową informację i musi wówczas przebywać sam w czytelni. Zakłada się, że operacje czytania i pisania nie trwają nieskończenie długo.

Problem ten jest abstrakcją problemu synchronizacji procesów korzystających ze wspólnej bazy danych. Wiele procesów naraz może odczytywać informację zawartą w bazie, ale tylko jeden może ją modyfikować. Zazwyczaj rozważa się trzy rozwiązania tego problemu, z których dwa dopuszczają zagłodzenie czytelników bądź pisarzy.

A. Rozwiązanie z możliwością zagłodzenia pisarzy

Zgodnie z warunkami zadania czytelnik, który zapragnie wejść do czytelni powinien móc to zrobić, jeśli jest ona pusta lub, jeśli są w niej już inni czytelnicy. W przeciwnym razie (tzn. gdy w czytelni jest pisarz) powinien być wstrzymany do czasu, gdy zajdą warunki

pozwalające mu wejść. Pisarz, który chce wejść do czytelni, powinien być wstrzymany do czasu, aż będzie ona pusta.

Przy takim postępowaniu jest możliwe jednak, że czytelnicy na zawsze zawładną czytelnią. Wystarczy, że w każdej chwili będzie w niej przebywał co najmniej jeden czytelnik, a pisarze nigdy nie doczekają się pustej czytelni. Możliwe jest tu zatem zagłodzenie pisarzy.

B. Rozwiązanie z możliwością zagłodzenia czytelników

Wydaje się, że jeśli pisarze chcą coś zapisać, to należy im to umożliwić najszybciej, jak jest to możliwe. Nie ma sensu, aby czytelnicy odczytywali przestarzałą informację. Zatem, jeśli jakiś pisarz czeka na wejście do czytelni, bo są w niej jeszcze czytelnicy, to nowo przybyły czytelnik powinien być. wstrzymany do czasu, gdy już żaden pisarz nie będzie czekał. Dzięki temu po pewnym czasie czytelnicy przebywający w czytelni w końcu ją opuszczą i będą mogli wejść do niej pisarze.

W tym przypadku jest możliwe jednak zagłodzenie czytelników przez intensywnie pracujących pisarzy. Wystarczy, że zawsze jakiś pisarz będzie czekał na możliwość pisania, a żaden z czytelników nie będzie mógł wejść do czytelni.

C. Rozwiązanie poprawne

Rozwiązanie poprawne powinno unikać zagłodzenia każdej z grup procesów. Najprościej jest wpuszczać do czytelni na przemian to czytelników, to pisarzy, np. zgodnie z kolejnością zgłoszeń, przy czym oczywiście pisarze muszą wchodzić pojedynczo, a czytelników można wpuszczać w grupach. Ponieważ jednak wszystkie operacje czytania mogą odbywać się jednocześnie, dlatego lepszym rozwiązaniem jest wpuszczenie do czytelni wraz z pierwszym czytelnikiem, na którego przyszła kolej, wszystkich innych czekających czytelników niezależnie od tego, czy mają jeszcze przed sobą pisarzy.

W niniejszej książce będziemy rozważać także rozwiązania, których żywotność (a więc brak możliwości zagłodzenia) wynika jedynie z faktu, że żywotny jest mechanizm synchronizacji, który w nich zastosowano.

O faktycznej kolejności wchodzenia do czytelni niewiele da się wówczas powiedzieć (por. 3.2.3, 7.2.3). W pewnym sensie będą to więc rozwiązania mniej sprawiedliwe, niż te opisane wyżej, choć równie poprawne. Będziemy zajmować się też pewnymi modyfikacjami problemu czytelników i pisarzy: z ograniczoną liczbą miejsc w czytelni (przykład 3.2.3) oraz z pisaniem poprzedzonym zgłoszeniem intencji (zad. 4.3.5).

2.6.3 Problem pięciu filozofów

Problem polega na zsynchronizowaniu działań pięciu filozofów, którzy siedzą przy okrągłym stole i myślą. Jest to ich główne zajęcie. Jednak od czasu do czasu każdy filozof głodnieje i aby móc dalej myśleć, musi się najeść. Przed każdym filozofem stoi talerz a pomiędzy kolejnymi dwoma talerzami leży jeden widelec. Na środku stołu stoi półmisek ze spaghetti. Jak wiadomo, spaghetti je się dwoma widelcami, więc aby rozpocząć jedzenie, filozof musi mieć do dyspozycji oba widelce. Podnosząc leżące przy nim widelce filozof uniemożliwia jedzenie swoim sąsiadom. Zakłada się jednak, że jeśli filozof podniósł już oba widelce, to w skończonym czasie naje się i odłoży je na stół. Rozwiązanie tego problemu musi gwarantować, że każdy filozof będzie mógł w końcu najeść się do syta. Dodatkowo zakłada się, że nie może ono wyróżniać żadnego z filozofów (algorytmy wszystkich pięciu filozofów dotyczące podnoszenia i odkładania widelców muszą być takie same).

Inny wariant tego problemu zakłada, że na stole stoi półmisek z rybami, które także trzeba jeść dwoma widelcami. Dla tych, którzy uważają, że zarówno rybę jak i spaghetti można zjeść jednym widelcem, najbardziej przekonujący będzie wariant z chińskimi filozofami używającymi pałeczek do jedzenia ryżu.

Problem pięciu filozofów jest abstrakcyjnym problemem, dla którego trudno znaleźć odpowiadającą mu rzeczywistą sytuację. Jest to jednak wzorcowy problem, na którego przykładzie pokazuje się zjawiska zagłodzenia i blokady. Zazwyczaj rozważa się trzy jego rozwiązania.

A. Rozwiązanie z możliwością blokady

Proste narzucające się rozwiązanie jest następujące. Zgłodniały filozof czeka, aż będzie wolny jego lewy widelec, podnosi go, a następnie czeka, aż będzie wolny prawy widelec i także go podnosi. Po najedzeniu się odkłada oba widelce na stół.

Z punktu widzenia pojedynczego filozofa jest to całkiem rozsądne postępowanie. Jednak ponieważ wszyscy oni postępują tak samo, łatwo wyobrazić sobie sytuację, choć bardzo mało prawdopodobną, w której każdy chwyci za swój lewy widelec i będzie czekać na to, aż jego prawy sąsiad skończy jeść i odłoży widelec. To jednak nigdy nie nastąpi, gdyż prawy sąsiad również czeka na to samo zdarzenie. Mamy tu typowy przykład zjawiska blokady, w której uczestniczą wszystkie współdziałające procesy.

B. Rozwiązanie z możliwością zagłodzenia

Poprzednie rozwiązanie można ulepszyć każąc filozofowi podnosić jednocześnie oba widelce wtedy, gdy tylko są wolne. Blokada w tym przypadku nie wystąpi, gdyż jeśli filozof podniesie już widelce, to w skończonym czasie odłoży je umożliwiając jedzenie sąsiadom.

Może się jednak tak zdarzyć, że jeden z filozofów będzie miał bardzo żarłocznych sąsiadów, którzy będą zajmować się głównie jedzeniem tak, że w każdej chwili przynajmniej jeden z nich będzie jadł. Jasne jest, że taki filozof nigdy nie doczeka się swoich widelców i prędzej czy później zostanie zagłodzony na śmierć. Tego typu postępowanie filozofów może być albo przypadkowe, albo wynikiem spisku mającego na celu zagłodzenie kolegi.

C. Rozwiązanie poprawne

Okazuje się, że sami filozofowie nie są w stanie rozwiązać problemu odżywiania się, jeśli będą obstawać przy tym, by żaden z nich nie był wyróżniony. (Dowód tego faktu można znaleźć w książce [Hare92, r. 10].) Potrzebny jest jakiś zewnętrzny arbiter rozstrzygający, którzy filozofowie będą mieli pierwszeństwo w spornych przypadkach. Arbitrem takim może być np. lokaj, który będzie dbał o to, aby w każdej chwili co najwyżej czterech filozofów konkurowało o widelce. Można łatwo wykazać, że wówczas przynajmniej jeden z nich będzie mógł podnieść oba widelce i jeść. Jeśli pięciu filozofów zapragnie jeść naraz, lokaj powstrzyma jednego z nich do czasu, aż któryś z pozostałych czterech skończy jeść, co zawsze kiedyś nastąpi. Rozwiązanie takie wyklucza zatem zarówno blokadę, jak i zagłodzenie któregoś z filozofów.

2.7 Mechanizmy niskopoziomowe

2.7.1 Przerwania

Do rozwiązania każdego z omówionych problemów potrzebny jest mechanizm umożliwiający wzajemne wykluczanie procesów. Okazuje się, że mechanizmy takie są dostępne na każdym komputerze. Każdy współczesny procesor jest wyposażony w mechanizm przerwań, natomiast każda pamięć jest zaopatrzona w arbiter pamięci. Ponadto w repertuarze instrukcji niektórych procesorów znajdują się instrukcje umożliwiające odczytanie, modyfikację i zapisanie zawartości komórki podczas jednego tylko dostępu do pamięci.

Mechanizm przerwań umożliwia przełączenie procesora na wykonywanie innego procesu. Przerwanie może być spowodowane albo jakimś zdarzeniem zewnętrznym, albo błędem wykonywania programu, albo wykonaniem specjalnej instrukcji powodującej przerwanie. W wyniku przerwania automatycznie zaczyna się wykonywać procedura zwana obsługą przerwania.

Mechanizm maskowania przerwań umożliwia zablokowanie następnych przerwań w czasie, gdy jest realizowana procedura obsługi poprzedniego. Do realizacji wzajemnego wykluczania można stosować przerwania powodowane wykonaniem specjalnej instrukcji i mechanizm ich maskowania.

W systemie jednoprocesorowym procesor wykonuje wszystkie procesy dzieląc swój czas pomiędzy nie oraz system operacyjny. System operacyjny ma najwyższy priorytet. Używając mechanizmu przerwań można tak napisać program współbieżny, aby sekcja krytyczna każdego procesu była wykonywana przez system operacyjny. Proces pragnący wejść do swojej sekcji krytycznej wykonuje instrukcję, która powoduje odpowiednie przerwanie, procesor przełącza się wówczas na wykonywanie systemu operacyjnego, a ten bez przeszkód (dzięki maskowaniu) wykonuje instrukcje obsługi przerwania będące sekcją krytyczną procesu, po czym procesor wraca do wykonywania procesów użytkowych. (W ten sposób realizuje się wzajemne wykluczania operacji semaforowych.)

Niestety programowanie przerwań wiąże się z ingerencją w obszar systemu operacyjnego i jest zastrzeżone tylko dla wąskiej grupy programistów systemowych. Zwykłym programistom nie można dać tego mechanizmu do ręki. Dodajmy jeszcze, że w systemie wieloprocesorowym mechanizm ten nie gwarantuje wzajemnego wykluczania przy dostępie do pamięci, gdyż dwa różne procesory mogą w tym samym czasie realizować różne przerwania.

2.7.2 Arbiter pamięci

Arbiter pamięci zapewnia wzajemne wykluczanie przy zapisie bądź odczycie pojedynczej komórki pamięci. Jeśli dwa procesory zapragną w jednej chwili sięgnąć do tej samej komórki pamięci, uzyskają do niej dostęp po kolei, choć kolejność ta nie jest z góry znana. Tak więc jeśli jeden procesor chce odczytać wartość komórki a drugi ją zmienić, to procesor czytający odczyta albo starą wartość, albo nową, ale nigdy nie będzie to np. połowa komórki stara a druga połowa nowa.

Algorytm Dekkera

Niebanalny algorytm wzajemnego wykluczania dwóch procesów oparty na własności gwarantowanej przez arbiter pamięci podał po raz pierwszy holenderski matematyk T. Dekker. Algorytm ten wymaga użycia nie jednej, jak by się wydawało, ale aż trzech różnych zmiennych.

W rozdziale 3 książki [BenA90] pokazano, w jaki sposób dochodzi się do tego algorytmu metodą kolejnych ulepszeń intuicyjnego rozwiązania. Prostszą wersję (choć nadal wymagającą trzech zmiennych) podał Peterson [PeteSI], a w książce [Hare92] można znaleźć

dowód poprawności tej wersji. Tam też znajdzie czytelnik algorytm będący uogólnieniem algorytmu Dekkera na wiele procesów. Okazuje się, że do synchronizacji N procesów potrzeba 2N-1 zmiennych, co oznacza, że liczba N musi być znana z góry, a przy każdej jej zmianie trzeba na nowo przepisywać odpowiednie fragmenty procesów. Poza tym algorytm wymaga od procesów aktywnego czekania. Z tych względów algorytm Dekkera okazał się niezbyt praktycznym narzędziem do realizacji wzajemnego wykluczania.

2.7.3 Instrukcje specjalne

Typowym przykładem instrukcji umożliwiającej odczytanie i zapisanie zawartości komórki pamięci podczas jednego dostępu jest instrukcja maszynowa Test&Set(X,R). Powoduje ona przepisanie zawartości komórki X do lokalnego rejestru R z jednoczesnym jej wy zerowaniem. Instrukcję tę można stosować do wzajemnego wykluczania dowolnej liczby procesów w następujący sposób. Na początku należy ustawić wartość wspólnie używanej zmiennej X na 1. Jedynka będzie oznaczać, że proces może wejść do swojej sekcji krytycznej. Przed wejściem do niej proces będzie wykonywał instrukcję:

```
R := 0; while R = O do Test&Set(X,R),
```

a po wyjściu będzie ponownie ustawiał wartość X na 1. Ponieważ tylko jeden proces będzie mógł stwierdzić, że ma w swoim rejestrze R wartość l, tylko jeden z nich będzie mógł wejść do swojej sekcji krytycznej.

Rozwiązane to ma jednak dwie wady. Po pierwsze, nie jest żywotne — żadnemu procesowi nie możemy zagwarantować, że w skończonym czasie uzyska prawo wejścia do sekcji krytycznej5. Po drugie, wymaga aktywnego czekania — proces czekający na wejście do swojej sekcji krytycznej jest nadal wykonywany zabierając cenny czas procesora. Mimo tych wad należy tu zauważyć, że właśnie za pomocą instrukcji typu Test&Set realizuje się wzajemne wykluczanie operacji semaforowych w systemach wieloprocesorowych ze wspólną pamięcią (por. [IsMa82, r. 10]).

Innymi instrukcjami maszynowymi tego typu są n p. SWAP(X,R), zamieniająca ze sobą zawartość X i R, oraz TSB(X,L), [Shaw78] równoważna wykonaniu instrukcji

```
if X then goto L else X := true.
```

Aktywne czekanie jest zjawiskiem, którego w programowaniu współbieżnym należy unikać. Wszystkie omawiane w następnych rozdziałach mechanizmy synchronizacji i komunikacji pozwalają go uniknąć udostępniając narzędzia do wstrzymywania procesów.

2.8 Mechanizmy wysokopoziomowe

2.8.1 Mechanizmy synchronizacji

Omawiane w poprzednim punkcie sprzętowe mechanizmy umożliwiające realizację wzajemnego wykluczania mają dość istotne wady. Algorytm Dekkera jest zależny od liczby synchronizowanych procesów, programowanie przerwań jest ingerencją w system operacyjny i, podobnie jak korzystanie z instrukcji specjalnych, wymaga programowania na niskim poziomie w języku asemblera.

Potrzeba stosowania języków wysokiego poziomu jest tak oczywista, że tym bardziej nie trzeba jej uzasadniać w odniesieniu do programowania współbieżnego. Języki wysokiego poziomu programowania współbieżnego muszą być wyposażone w mechanizmy umożliwiające synchronizację procesów i komunikację między nimi.

Semafory

Pierwszym mechanizmem językowym wysokiego poziomu służącym do synchronizacji procesów były wprowadzone przez Dijkstrę semafory [DijkGS].

Nazwa semafor nie bez powodu może kojarzyć się z urządzeniem powszechnie stosowanym na kolei. Jego zadaniem jest bowiem przepuszczanie i zatrzymywanie procesów. W rzeczywistości jest to zmienna całkowita, na której wolno wykonywać tylko dwie operacje odpowiadające podnoszeniu i opuszczaniu semafora kolejowego.

W sieci Ethernet stosuje się działający na podobnej zasadzie algorytm dostępu do medium transmisji. Algorytm ten jednak wymaga, aby po nieudanej próbie proces odczekał pewien losowy czas, dzięki czemu prawdopodobieństwo zagłodzenia zbiega do zera. Choć być może do ich realizacji trzeba użyć aktywnego czekania na niższym poziomie.

Niestety podstawową wadą semaforów jest to, że ich używanie prowadzi zazwyczaj do niestrukturalnego programowania. Okazało się też, że w niektórych typowych przypadkach programowanie za pomocą klasycznego semafora Dijkstry staje się bardzo skomplikowane. Dlatego szybko powstały różne uogólnienia operacji semaforowych.

Semafor klasyczny i niektóre jego rozszerzenia są omówione w rozdz. 3. Tam też proponujemy kilka zadań polegających na implementacji omówionych rozszerzeń za pomocą semafora klasycznego. Czytelnik będzie się mógł sam przekonać, jak trudno jest programować za pomocą tego narzędzia. W rozdziale 8 omawiamy semafory w systemie Unix. Są one bardzo mocnym narzędziem programowania współbieżnego, zawierają bowiem w sobie większość proponowanych rozszerzeń semafora klasycznego. Jak pokażemy, za ich pomocą można bardzo zwięźle zapisać rozwiązania niektórych niebanalnych problemów. (Należy tu też dodać, że uzyskano to częściowo dzięki zastosowaniu specjalnej notacji ukrywającej szczegóły niezbyt czytelnego programowania współbieżnego w systemie Unix.)

Monitory

Niestrukturalne programowanie za pomocą semaforów było przyczyną wielu trudnych do wykrycia błędów. Podejmowano więc próby wprowadzenia mechanizmu umożliwiającego strukturalne programowanie synchronizacji. Pierwszymi takimi mechanizmami były regiony krytyczne i warunkowe regiony krytyczne zaproponowane przez P. Brinch Hansena [Brin78, IsMa,82]. Z prób tych zrodził się w końcu mechanizm monitora, który niezależnie od siebie po raz pierwszy zaproponowali Brinch Hansen [Brin74] i Hoare [Hoar74].

Nazwa monitor nawiązuje tu do urządzenia zajmującego się śledzeniem i nadzorowaniem pracy, jego głównym przeznaczeniem jest bowiem zarządzanie wybranym zasobem komputera. Monitory stały się podstawowym narzędziem synchronizacji kilku języków programowania współbieżnego. Języki te różnią się jednak szczegółami implementacji tego mechanizmu. W rozdziale 4 omawiamy dokładnie klasyczny mechanizm monitora oraz podajemy sposoby jego zapisywania w różnych językach programowania.

Inne mechanizmy

Oprócz różnych odmian semaforów istnieją także inne niestrukturalne mechanizmy synchronizacji, takie jak numeratory i liczniki zdarzeń (sequencers and eventcounts) (por.

[Ma0087, WeisOOb, Gosc91]). Wśród strukturalnych mechanizmów warto wspomnieć o wyrażeniach .ścieżkowych (por. [CaHa74, IsMa82, MaOO87, BiSh88]). W niniejszej książce nie będziemy się jednak nimi zajmować.

2.8.2 Mechanizmy komunikacji i synchronizacji

Zarówno monitory jak i semafory są mechanizmami służącymi głównie do synchronizacji procesów. W tym punkcie omawiamy mechanizmy umożliwiające przesyłanie komunikatów między procesami i zapewniające właściwą ich synchronizację.

Mechanizmy komunikacji używane w językach programowania przypominają bardzo sposoby, jakimi my sami komunikujemy się ze sobą. Każdy z nas jest przecież czymś w rodzaju procesora mającego dostęp do własnej pamięci (tej operacyjnej w mózgu i tej pomocniczej w książkach i notatnikach) i każdy z nas musi komunikować się z innymi ludźmi.

Spotkania

Najbardziej bezpośredni sposób komunikacji między dwojgiem ludzi to osobiste spotkanie. Spotkania jako mechanizm komunikacji między procesami zaproponował w roku 1978 C. A. R. Hoare w języku o nazwie CSP (Communicating Sequential Processes} [Hoar78]. Spotkanie w CSP przypomina spotkanie dwóch dobrze znających się osób w z góry umówionym miejscu, podczas którego jedna osoba coś daje, a druga to odbiera. Można powiedzieć, że jest to spotkanie symetryczne (rozdz. 5) ze względu na relację znajomości (choć jednocześnie asymetryczne ze względu na kierunek przepływu informacji).

Ideę spotkań zaproponowanych w CSP rozwinięto podczas projektowania języka Ada [Pyle86]. Spotkanie w Adzie przypomina wizytę klienta w zakładzie naprawczym. Jest to spotkanie asymetryczne (rozdz. 6) zarówno ze względu na relację znajomości, jak i ze względu na aktywność podczas spotkania. Klient wie do kogo przyszedł, ale jego rola w tym spotkaniu ogranicza się do czekania na wykonanie usługi. Pracownik zakładu nie zna klienta, za to on podczas spotkania działa. Tego typu spotkanie umożliwia wymianę albo w obie strony (klient przynosi zepsutą rzecz, pracownik dokonuje naprawy i zwraca klientowi naprawioną rzecz), albo tylko w jedną stronę (naprawy nie można wykonać na miejscu i trzeba umówić się na nowe spotkanie).

Przestrzeń krotek

Spotkanie się z procesem lub wysianie do niego komunikatu wymaga znajomości nazwy procesu. Jednak są przypadki, w których, aby działać, nie trzeba w ogóle znać dostawców informacji ani odbiorców przygotowanych przez nas danych. Cala wiedza dotycząca n p. matematyki została zapisana w tysiącach tomów książek i czasopism i jest dostępna każdemu, kto chce się zajmować tą dziedziną nauki. Uczony matematyk dowodząc nowe twierdzenie korzysta z dorobku wielu innych matematyków. Nie musi przy tym ani znać ich nazwisk, ani wiedzieć, czy jeszcze żyją. Podobnie uczony ten nie wie, kto i kiedy skorzysta z jego nowego twierdzenia.

Ten specyficzny sposób komunikacji nie zakładający żadnych powiązań między procesami, ani w czasie, ani w przestrzeni, został zaproponowany w języku Linda [Gele85, AhCGSG, BenAOO]. Podstawą komunikacji między procesami jest w nim przestrzeń krotek (rozdz. 7), która zawiera wszystkie wyniki działania procesów. Wyniki te są umieszczane przez procesy w postaci ciągów obiektów różnych typów. Procesy mogą z nich korzystać dzięki

mechanizmowi selektywnego wyboru umożliwiającemu sięganie do przestrzeni krotek tylko po tę informację, która jest potrzebna.

Potoki

Tak jak semafor wymyślono, by maksymalnie uprościć rozwiązanie problemu wzajemnego wykluczania, tak pojęcie potoku stworzono, by uprościć rozwiązanie problemu producenta i konsumenta. Potok jest dla procesów czymś w rodzaju bufora, do którego można wstawiać komunikaty i z którego można je pobierać. W rzeczywistości jest to plik, na którym wiele procesów może jednocześnie wykonywać operacje pisania i czytania odpowiadające wkładaniu do bufora i wyjmowaniu z niego.

Potoki po raz pierwszy użyto w systemie operacyjnym Unix (rozdz. 9). Ich szczególna postać, potoki nienazwane, umożliwia utożsamienie wyjścia jednego procesu z wejściem innego. Dzięki nim można więc tworzyć całe ciągi współpracujących procesów, kolejno przetwarzających informacje na podobnej zasadzie jak wykonuje się montaż na taśmie produkcyjnej. Tego typu przetwarzanie nazywa się przetwarzaniem potokowym. Dość niefortunnie ciąg procesów połączonych potokami nazywa się także potokiem. Mechanizm potoków jest dostępny także w systemie MS-DOS (rozdz. 11).

Komunikaty i kanały

Potoki umożliwiają jednokierunkowe przekazywanie informacji. W nowszych wersjach systemu Unix wprowadzono mechanizm komunikatów przekazywanych przez specjalne kanały (rozdz. 9), dzięki któremu można przesyłać między procesami informacje o dowolnej strukturze i w obie strony. W kanałach wyróżnia się podkanały, z których każdy pełni rolę odrębnego bufora. W rezultacie kanał przypomina szafę z przegródkami, w której można zostawiać różne wiadomości różnym osobom. Potok jest szczególnym przypadkiem kanału, który ma tylko jeden podkanał i przez który przesyła się jedynie ciąg bajtów.

Zdalne wywalanie procedury

Asymetryczne spotkania użyto także w mechanizmie językowym zwanym zdalnym wywalaniem procedury lub w skrócie RPC (Remote Procedure Call) (rozdz. 10).

Proces potrzebujący wykonania pewnej usługi przez inny proces wywołuje jego wewnętrzną procedurę wysyłając, mu odpowiednie parametry, po czym czeka na wynik. Mechanizm zdalnego wywołania procedury jest powszechnie używany w rozproszonych systemach operacyjnych.

Inne mechanizmy

Potoki i zdalne wywołanie procedury to właściwie już standard, ale jest wiele innych mechanizmów umożliwiających komunikację między procesami na poziomie systemu operacyjnego, np. skrzynki pocztowe (mailboxes, mailslots), łącza (links), porty (ports), gniazda (sockets), rozgłaszanie (broadcasting), sygnały (signals). Ich definicje mogą być jednak różne w różnych systemach (por. np. [Gosc91]). O niektórych wspominamy przy okazji omawiania systemu Unix. Mechanizmy te wspomagają realizację językowych mechanizmów programowania współbieżnego.

2.8.3 Klasyfikacja

Istniejące mechanizmy synchronizacji i komunikacji można podzielić w zależności od architektury sprzętu, na którym się je zazwyczaj stosuje, na mechanizmy przeznaczone dla systemów scentralizowanych i rozproszonych.

Przez system scentralizowany rozumiemy taką architekturę systemu komputerowego, w której wszystkie procesory mają dostęp do jednej wspólnej pamięci. Oznacza to, że procesy mogą komunikować się i synchronizować przekazując sobie potrzebne informacje przez wspólną pamięć. Mechanizmami realizowanymi we wspólnej pamięci są semafory, monitory, potoki, komunikaty unixowe oraz przestrzeń krotek.

Przez system rozproszony rozumiemy taką architekturę systemu komputerowego, w której każdy procesor ma dostęp tylko do swojej własnej pamięci, natomiast wszystkie procesory połączone są łączami umożliwiającymi przesyłania między nimi danych. Nie ma przy tym żadnych założeń co do rodzaju, długości i szybkości tych łączy. W tym rozumieniu systemem rozproszonym może być zarówno płyta zawierająca kilka transputerów, lokalna sieć złożona z wielu komputerów osobistych, jak i sieć rozległa obejmująca różnego typu komputery rozmieszczone w różnych miastach i państwach, a nawet kontynentach. Mechanizmami przeznaczonymi do komunikacji i synchronizacji w systemach rozproszonych są spotkania oraz RPC.

Jak już pisaliśmy, programowanie współbieżne z przeznaczeniem dla systemów rozproszonych nazywa się czasem programowaniem rozproszonym, aby podkreślić w ten sposób odmienny charakter powiązań między procesami.

Przedstawiony podział nie jest jednak ostateczny. Spotkania i RPC można stosować w systemach scentralizowanych (co prawda są one mniej efektywne niż mechanizmy zaprojektowane specjalnie dla tych systemów). Z drugiej strony, takie mechanizmy jak potoki i komunikaty unixowe logicznie niczym nie różnią się od mechanizmu zwykłych komunikatów przesyłanych w systemach rozproszonych. Możliwe jest także zrealizowanie rozproszonej przestrzeni krotek.

Podział na scentralizowane i rozproszone mechanizmy programowania wygląda nieco inaczej z punktu widzenia struktury programu. Jeśli przyjmiemy, że program jest scentralizowany, gdy procesy komunikują się przez zmienne globalne, a rozproszony, gdy procesy nie mają dostępu do zmiennych globalnych i jedynym sposobem wymiany informacji jest wysłanie komunikatu, to za mechanizmy scentralizowane musimy uznać mechanizmy synchronizacji (semafory, monitory) a za mechanizmy rozproszone — mechanizmy komunikacji (potoki, spotkania, przestrzeń krotek).

Z drugiej strony zakładając, że sygnał synchronizujący jest po prostu pustym komunikatem, możemy uznać, że wszystkie wspomniane tu mechanizmy są mechanizmami komunikacji. Mechanizmy te można podzielić w zależności od narzucanych przez nie powiązań między procesami w czasie (kto na kogo musi czekać) oraz od powiązań w przestrzeni (kto z kim się komunikuje).

Ze względu na powiązania czasowe mechanizmy komunikacji dzielimy na synchroniczne i asynchroniczne. Z komunikacją synchroniczną mamy do czynienia wówczas, gdy nadawca komunikatu musi poczekać, aż odbiorca go odbierze. Inaczej mówiąc, nadawca i odbiorca synchronizują się ze sobą w chwili przekazywania komunikatu. Wszelkiego typu spotkania (w tym także RPC) są zatem mechanizmami komunikacji synchronicznej.

Z komunikacją asynchroniczną mamy do czynienia wówczas, gdy nadawca może wysłać komunikat w dowolnej chwili bez względu na to, co wtedy robi odbiorca. Nadawanie komunikatu nie blokuje zatem nadawcy, ale oczywiście odbiór komunikatu jest możliwy dopiero po jego nadaniu. Synchronizacja jest więc tu tylko jednostronna — odbiorca musi

dostosować się do nadawcy, ale nie odwrotnie. Poza spotkaniami wszystkie inne omawiane tu mechanizmy zapewniają komunikację asynchroniczną.

Wskazując bezpośrednio lub pośrednio partnerów komunikacji określa się powiązania procesów w przestrzeni. Sposoby powiązania można podzielić w zależności od tego, czy nadawca komunikatu jest z góry znany, czy nieznany odbiorcy, oraz czy odbiorca jest z góry znany, czy nieznany nadawcy. Składnia spotkań w CSP wymaga jawnego wskazania zarówno nadawcy, jak i odbiorcy komunikatu. Składnia spotkań asymetrycznych (Ada, RPC) wymaga jawnego wskazania tylko jednego z uczestników komunikacji. Pozostałe mechanizmy wymagają wskazania pośrednika w komunikacji (potoku, kanału, semafora, monitora). Jednak takie mechanizmy, jak kanały w języku occam czy potoki nienazwane w Unixie, wiążą zawsze tylko jedną parę nadawca-odbiorca, zatem w tych przypadkach wskazując pośrednika wskazuje się jednoznacznie partnera. Uznamy, że nadawca rzeczywiście nie zna odbiorcy, tylko wtedy, gdy jego komunikat może odebrać jeden z wielu możliwych odbiorców. Odbiorca nie zna nadawcy wtedy, gdy komunikat, którego się spodziewa, może nadejść od jednego z wielu nadawców. Omówione podziały ilustruje tablica 2.1.

TABLICA 2.1 Podział mechanizmów komunikacji

Nadawca	Odbiorca	Synchroniczne (stosowane w systemach rozproszonych)	Asynchroniczne (stosowane w systemach scentralizowanych)
Znany	znany	spotkania w CSP	potoki nienazwane
znany	nieznany	spotkania w Adzie	potoki, kanały unixowe
nieznany	znany	spotkania w Adzie	przestrzeń krotek
nieznany	nieznany	spotkania w Edipie	semafory, monitory

W niniejszej książce przyjęliśmy następujący układ. Najpierw omawiamy ogólne mechanizmy służące przede wszystkim do synchronizacji (semafory i monitory), a następnie mechanizmy komunikacji synchronicznej (spotkania) i mechanizmy komunikacji asynchronicznej (przestrzeń krotek). Na końcu zajmujemy się wybranymi mechanizmami dostępnymi na poziomie systemu operacyjnego Unix, takimi jak semafory, komunikaty i zdalne wywołanie procedury. W ostatnim rozdziale omawiamy mechanizm potoków w systemie MS-DOS.

2.8.4 Rodzaje programów współbieżnych

W zależności od sposobu komunikacji między procesami wyróżnia się kilka typów programów współbieżnych [AnclrOl]. Wspomnieliśmy już o przetwarzaniu potokowym. Każdy proces w programie realizującym takie przetwarzanie pobiera dane tylko z jednego źródła (procesu lub kanału), przetwarza je i wysyła do jednego odbiorcy (procesu lub kanału). Takie procesy nazywa się filtrami. Przykłady przetwarzania potokowego zawierają zadania I LIII.8. Bardziej złożoną strukturę programu w postaci acyklicznego grafu uzyskuje się wtedy, gdy poszczególne procesy pobierają dane z kilku źródeł i wysyłają wyniki do kilku odbiorców (por. 5.3.13, w którym omówiono potok dwuwymiarowy).

Inną grupę stanowią programy współbieżne złożone z wielu identycznie działających procesów. Zasługują one na szczególną uwagę, gdyż zwykle łatwiej je tworzyć, rozumieć i modyfikować. Struktura powiązań między procesami w takim programie zależy od rozwiązywanego zadania (por. 5.3.15). W skrajnym przypadku komunikaty mogą być

przekazywane między każdą parą takich procesów. Istnieje kilka sposobów zorganizowania komunikacji między identycznie działającymi procesami. Najważniejsze to:

- rozgłaszanie (broadcasting),
- bicie serca (heartbcating),
- przekazywanie żetonu (token passing),
- wspólny worek (common bag).

Rozgłaszanie to rozsyłanie takiego samego komunikatu do wszystkich procesów (por. algorytm Ricarta i Agrawali w 5.2.1 oraz zadania 5.3.5, 5.3.6).

Bicie serca jest techniką stosowaną wtedy, gdy procesy mogą komunikować się bezpośrednio tylko z niewielkim podzbiorem wszystkich procesów. Polega ona na wielokrotnym wykonaniu następujących czynności: wysłanie informacji do procesów sąsiednich (tzn. bezpośrednio połączonych łączami komunikacyjnymi), odebranie od nich odpowiedzi, przetworzenie tych odpowiedzi i ponowne rozesłanie informacji. Czynności te powtórzone odpowiednią liczbę razy powodują, że cały układ procesów osiąga pewien zamierzony stan (por. 5.3.15).

Przekazywanie żetonu jest częstą techniką stosowaną głównie do synchronizacji procesów w systemach rozproszonych. Żeton jest pewnym wyróżnionym komunikatem kolejno przekazywanym między procesami tworzącymi zamknięty pierścień. Każdy z procesów może odczytywać informację zawartą w żetonie i odpowiednio ją zmieniać, dzięki czemu jest informowany i sam informuje innych o stanie systemu. Czasami wystarczy fakt otrzymania żetonu (por. przekazywanie uprawnień w 5.2.1).

Wspólny worek zawiera zbiór komunikatów z opisem i parametrami działań do wykonania. Każdy proces pobiera taki komunikat, wykonuje działanie i wynik wkłada z powrotem do worka. Wynik ten może być wynikiem ostatecznym lub wymagać dalszego przetwarzania. Wyróżniony proces wkłada do worka komunikaty inicjujące i pobiera z niego wyniki (por. 7.3.7, 7.3.8, 7.3.9).

Innym, bardzo często spotykanym sposobem komunikacji między procesami jest układ klient-proces obsługujący (sender). Klient to proces wysyłający żądania, które wyzwalają reakcję procesu obsługującego. Klient ma w tym układzie inicjatywę, żądania są bowiem wysyłane w dowolnej wybranej przez niego chwili. Po wysłaniu żądania klient zazwyczaj czeka na reakcję procesu obsługującego. Proces obsługujący jest zwykle procesem nieskończonym, który cyklicznie czeka na żądanie od klienta, a, następnie obsługuje je. Istnieje kilka wariantów takiego modelu komunikacji między procesami:

- jeden klient —jeden proces obsługujący (por. 3.3.10, 5.3.3),
- wielu klientów — jeden proces obsługujący (por. 5.3.7, 5.3.8),
- wielu klientów — wiele procesów obsługujących (por. 10.3.6, 10.3.7),
- hierarchia — proces obsługujący niższego poziomu jest klientem procesu wyższego poziomu (por. 4.3.17, 5.3.16, 10.3.7),
- interakcja — proces obsługujący i klient zamieniają się rolami w zależności od etapu obliczeń (por. 6.3.2).

W bardziej złożonych programach współbieżnych procesem obsługującym może być cały potok procesów (por. 5.3.4, 5.3.10) lub układ identycznych procesów komunikujących się przez rozgłaszanie, przekazywanie żetonu (por. 5.3.2) czy na zasadzie bicia serca.

Należy zauważyć, że podana tu klasyfikacja programów współbieżnych nie ma nic. wspólnego z klasyfikacją mechanizmów programowania. Ten sam schemat komunikacji można zrealizować za pomocą różnych mechanizmów. Należy jednak podkreślić, że pewne

mechanizmy są szczególnie przydatne do realizowania określonych typów programów, np. potoki do przetwarzania potokowego, asynchroniczne spotkania do układu klient-proces obsługujący, przestrzeń krotek do przetwarzania na zasadzie wspólnego worka.

2.8.5 Notacja

We wszystkich podawanych w tej książce rozwiązaniach przyjęliśmy następującą konwencję notacyjną. Nazwy elementów uczestniczących w synchronizacji i komunikacji (procesów, semaforów, monitorów, procedur eksportowanych, wejść, kanałów oraz komunikatów przesyłanych między procesami) będziemy zapisywać wielkimi literami. Wielkimi literami będziemy oznaczać także stałe (określające zazwyczaj liczbę procesów). Nazwy zmiennych prostych i tablic będziemy zapisywać małymi literami. W celu uproszczenia zapisu procesów pewne ciągi instrukcji nieistotne z punktu widzenia problemu synchronizacji czy komunikacji będziemy zastępowali identyfikatorami, które można uważać za zwykłe wywołania lokalnych procedur procesu.

W rozdziałach poświęconych dostępnym w systemie Unix narzędziom programowania współbieżnego i rozproszonego (r. 8, 9 i 10) posługujemy się abstrakcyjną notacją, wprowadzoną specjalnie na potrzeby tej książki. Jest ona zgodna z notacją iiżywaną w innych rozdziałach, dzięki czemu wszystkie prezentowane algorytmy mogą być zapisywane w jednolitej postaci. Podajemy jednak również sposób przetłumaczenia tej notacji na zapis w języku C, powszechnie stosowanym przy pisaniu programów wykonywanych w systemie Unix. Zapis programów współbieżnych w języku C jest z reguły mało czytelny, abstrakcyjna zwięzła notacja zwiększy przejrzystość prezentowanych algorytmów i ułatwi ich zrozumienie. Niektóre z algorytmów zostaną dodatkowo zapisane w języku C, w postaci gotowych do wykonania programów. Jednak w celu zapewnienia większej czytelności oraz oszczędności miejsca, prezentowany kod jest pozbawiony wszelkich szczegółów zbędnych do zrozumienia opisywanego mechanizmu. Na przykład zaniedbujemy sprawdzanie kodu powrotu przy wywołaniu funkcji systemowych oraz z reguły pomijamy pliki definicyjne (nagłówkowe).

Opis przedstawianych mechanizmów unixowych nie jest kompletny. W przypadku semaforów i komunikatów pomijamy pewne drobiazgi, natomiast dla zdalnego wywołania procedur przedstawiamy tylko funkcje tworzące najwyższy poziom tego mechanizmu. Oprócz nich jest dostępnych kilkadziesiąt innych funkcji, nie dają one jednak wielu nowych możliwości programowania rozproszonego. Zainteresowanego czytelnika odsyłamy do dokumentacji systemowej. Należy pamiętać, że niniejsza książka nie jest podręcznikiem programowania w systemie Unix i przy uruchamianiu własnych programów trzeba aktywnie korzystać z dokumentacji systemowej.

Wszystkie prezentowane programy zapisane w C były testowane. Należy jednak pamiętać, że teksty programów mogą zależeć od wersji systemu, w którym są uruchamiane (np. nazwy funkcji, liczby argumentów funkcji), oraz od konkretnej instalacji (np. ścieżki w dyrektywach #include).

Więcej informacji na temat omawianych w tym rozdziale pojęć można znaleźć np. w książkach [BenASO, BenAOO, IsMa<32, MaO087, SiPG91, Brin79, Shaw79].

3. Semafory

3.1 Wprowadzenie

3.1.1 Semafor jako abstrakcyjny typ danych

Pierwszym mechanizmem synchronizacyjnym stosowanym w językach wysokiego poziomu były semafory wprowadzone przez Dijkstrę [Dijk68]. Zdefiniowany przez niego abstrakcyjny typ danych wraz z operacjami umożliwiającymi wstrzymywanie i wznawianie procesów będziemy w tej książce oznaczać angielskim słowem semaphore, natomiast semaforami będziemy nazywać zmienne tego typu.

Na semaforze, oprócz określenia jego stanu początkowego, można wykonywać tylko dwie operacje:

- podniesienie semafora,
- opuszczenie semafora,

o których zakłada się, że są niepodzielne, co oznacza, że w danej chwili może być wykonywana jedna z nich.

W zależności od sposobu zdefiniowania tych operacji wyróżnia się kilka rodzajów semaforów. Dijkstra zaproponował dwa; ogólny i binarny. Inni autorzy zaproponowali m.in. semafor uogólniony l dwustronnie ograniczony.

3.1.2 Semafor ogólny

Definicja klasyczna

Semafor ogólny jest po prostu zmienną całkowitą z dwoma wyróżnionymi operacjami. Zgodnie z klasyczną definicją podaną przez Dijkstrę, podniesienie semafora S to wykonanie instrukcji

```
5 := 5 + l
```

a jego opuszczenie to wykonanie instrukcji

```
czekaj, aż S > 0; S := S – 1.
```

Jest rzeczą dyskusyjną, czy taki sposób zdefiniowania operacji podniesienia i opuszczenia spełnia warunek, aby operacje semaforowe były niepodzielne. Jeśli proces rozpoczynający opuszczanie semafora stwierdzi, że jego wartość nie jest dodatnia, musi zaczekać. Ale aby mógł on zakończyć opuszczanie semafora, jakiś inny proces musi mieć możliwość wykonania operacji podniesienia. W rzeczywistości w takim przypadku operacja opuszczania musi być przerwana w trakcie sprawdzania warunku `9 > 0`. Podczas tej przerwy jakiś inny proces może rozpocząć opuszczanie semafora i ta operacja także musi być przerwana. W rezultacie można mieć, wiele rozpoczętych a nie zakończonych operacji opuszczania. Jednak w chwili, gdy jakiś proces zwiększy ,9, tylko jedna z tych operacji będzie mogła wykonać się do końca, ponieważ zakłada się, iż stwierdzenie S > 0 i wykonanie `S := S – l`, to jedna niepodzielna operacja. W

definicji klasycznej przyjmuje się więc, że operacja opuszczenia semafora jest niepodzielna tylko wtedy, gdy sprawdzany w niej warunek jest spełniony.

Definicja „praktyczna "

W niniejszej książce przyjmiemy definicję, która została podana przez BenAriego [BenA89]. Jest ona zgodna z intuicją i nie kryje opisanych wyżej subtelności. Według niej opuszczenie semafora to wykonanie instrukcji:

- `jeśli S > O, to S := S — 1, w przeciwnym razie wstrzymaj działanie procesu wykonującego tę operację.;`

a podniesienie semafora to wykonanie instrukcji:

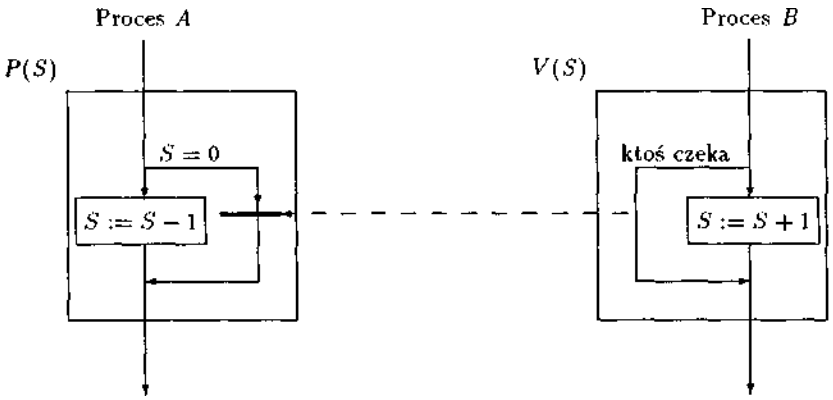
- `jeśli są procesy wstrzymane w wyniku wykonania operacji opuszczania semafora S, to wznów jeden z nich, w przeciwnym razie S := S + 1.`

Schemat synchronizacji za pomocą tak zdefiniowanego semafora przedstawia rys. 3.1.

Ponieważ operacje podniesienia i opuszczenia semafora są niepodzielne, w danej chwili tylko jeden proces może badać stan semafora S i ewentualnie zwiększyć lub zmniejszyć jego wartość.

Zgodnie z definicją „praktyczną" podniesienie semafora w chwili, gdy czekają na to jakieś inne procesy, powoduje, że któryś z nich na pewno będzie wznowiony. Definicja klasyczna tego nie zapewnia. Zgodnie z nią proces, który podniesie opuszczony semafor, może zaraz potem sam wykonać operację opuszczania i stwierdzić przed innymi procesami, że wartość semafora jest dodatnia, po czym zmniejszyć ją o jeden.

O sposobie wyboru procesu, który zostanie wznowiony, zakłada się jedynie, że jest to sposób nie powodujący zagłodzenia żadnego z czekających procesów. (Warunek ten spełnia np. wybór zgodny z kolejnością zgłoszeń, przy programowaniu jednak nie można zakładać, że w ten właśnie sposób rzeczywiście jest realizowana kolejka procesów oczekujących na podniesienie semafora.)



RYS. 3.1. Zasada działania operacji semaforowych

Notacja

Semafor ogólny będziemy deklarować jako obiekt typu semaphore. Wartość początkową semafora będziemy określać w miejscu jego deklaracji. Tak więc

```
var SEM: semaphore := 4;
```

jest deklaracją semafora SEM o wartości początkowej 4. Podobny sposób inicjowania przyjmiemy także dla zmiennych innych typów.

Dijkstra operację podnoszenia oznaczył literą V od pierwszej litery holenderskiego słowa vrimakenn verhogen, co znaczy zwolnić lub zwiększyć, a operację opuszczania literą P od holenderskiego słowa passeren lub proberen, co znaczy przejść lub próbować. V(5) oznacza zatem podniesienie semafora 5, a P(S) jego opuszczenie.

W literaturze na określenie operacji podniesienia i opuszczenia używa się także słów wait i signal. Tych samych słów używa się jednak na oznaczenie opisanych w następnym rozdziale operacji na zmiennych typu condition. Aby uniknąć ewentualnych nieporozumień, pozostaniemy przy oznaczeniach P i V.

3.1.3 Semafor binarny

W odróżnieniu od semafora ogólnego semafor binarny nie jest zmienną całkowitą lecz zmienną logiczną albo, inaczej mówiąc, pojedynczym bitem. Semafor binarny może zatem przyjmować tylko wartości 0 lub 1 (true lub false).

Definicja klasyczna

Według definicji klasycznej podnoszenie semafora binarnego to wykonanie instrukcji:

- `S := l`

a jego opuszczenie to wykonanie instrukcji

- `czekaj, aż S = 1; S := 0.`

Z przyczyn, które już omówiliśmy, przyjmiemy w tej książce definicję „praktyczną".

Definicja „praktyczna "

Według definicji „praktycznej" operacji opuszczenia semafora binarnego odpowiada wykonanie instrukcji:

- `jeśli S = 1, to S := 0, w przeciwnym razie wstrzymaj działanie procesu wykonującego tę operację;`

a operacji podniesienia — instrukcja:

- `jeśli są procesy wstrzymane w wyniku wykonania operacji opuszczania semafora S, to wznów jeden z nich, w przeciwnym razie S := 1.`

Notacja

Semafor binarny, ze względu na swą prostotę, jest łatwiejszy do realizacji niż semafor ogólny (przekonują o tym odpowiednie przykłady w dalszych rozdziałach). Będziemy podkreślać, wyjątkowość semafora binarnego dodając przy deklaracji słowo binary. Operację podnoszenia semafora binarnego będziemy oznaczać literami VB a opuszczania - literami PB. W istocie większość semaforów używanych w przykładach i rozwiązaniach zadań w tym rozdziale to semafory binarne.

Związek między semaforem binarnym a ogólnym

Semafor binarny nie jest szczególnym przypadkiem semafora ogólnego, nie pamięta on bowiem liczby wykonanych na nim operacji podnoszenia. W wielu typowych zastosowaniach semafor binarny można jednak zastąpić semaforem ogólnym. Jest tak n p. wtedy, gdy semafor binarny służy do realizacji wzajemnego wykluczania. Ma wówczas wartość początkową równą l, operacja PB poprzedza wejście do sekcji krytycznej, a operacja VB następuje po wyjściu z niej. Taki sposób korzystania z semafora binarnego gwarantuje, że nigdy nie będą na nim wykonywane dwie bezpośrednio po sobie następujące operacje VB. Semafor binarny można wówczas zastąpić semaforem ogólnym, a operacje PB i VB operacjami P i V. Jeśli jednak pozwolimy na wielokrotne wykonywanie operacji VB na semaforze binarnym, to takiego semafora nie można zastąpić semaforem ogólnym.

Za pomocą semafora binarnego można natomiast symulować semafor ogólny. Do symulacji jednego semafora ogólnego potrzeba dwóch semaforów binarnych i jednej zmiennej całkowitej. Symulacja taka jest przedmiotem zadania 3.3.1.

Podnoszenie semafora binarnego

Kolejne wielokrotne wykonanie operacji VB na semaforze binarnym ma taki sam skutek, jak pojedyncze wykonanie tej operacji. Choć własność ta może być w niektórych zastosowaniach przydatna, może też bardzo utrudnić wykrycie błędów w programie. To, w jakiej kolejności procesy wykonają operacje PB i VB, może zależeć w rzeczywistości od prędkości procesorów lub sposobu szeregowania procesów do pojedynczego procesora. Zauważmy, że ponieważ semafor binarny nie pamięta liczby wykonanych operacji VB, więc np. wykonanie w kolejności PB(S], VB(S), VB(S), PB(S) daje zupełne inny wynik niż wykonanie w kolejności PB(S), VB(S), PB(S), VB(S), podczas gdy kolejność ta może zależeć od wzajemnych prędkości procesów wykonujących drugą operację PB i drugą operację VB. Aby uniknąć takiej niejednoznaczności, założymy, że podnoszenie podniesionego semafora binarnego jest błędem wykonania powodującym natychmiastowe zakończenie programu współbieżnego.

3.1.4 Rozszerzenia i modyfikacje

Semafor dwustronnie ograniczony

Operacje P i V na semaforze ogólnym nie są symetryczne — operacja P może spowodować wstrzymanie procesu, operacja V nie wstrzymuje procesu. Jednym z rozszerzeń pojęcia semafora jest semafor dwustronnie ograniczony [Lipt74, MaO087], czyli taki, który z jednej strony nie może przyjmować wartości ujemnych, ale z drugiej strony nie może przekroczyć pewnej dodatniej wartości N. Operacja podnoszenia semafora VD jest więc w tym przypadku symetryczna do operacji opuszczania PD:

- jeśli S = O (S = N) to wstrzymaj działanie procesu wykonującego tę operację, w przeciwnym razie jeśli są procesy wstrzymane w wyniku podnoszenia (opuszczania) semafora S, to wznów jeden z nich, w przeciwnym razie S := S – l (S := S + 1).

(Zauważmy, że nieograniczoność klasycznego semafora jest teoretyczna. W każdej realizacji istnieje ograniczenie górne na jego wartość. Jest ono jednak na tyle duże, że program, który mógłby je przekroczyć, z pewnością będzie błędny.)

Semafor uogólniony

Innym rozszerzeniem pojęcia semafora jest dopuszczenie, aby był on zmieniany nie o l, ale o dowolną wskazaną liczbę naturalną [Vant72, MaOO87]. Operacje podnoszenia (VG) i opuszczania semafora (PG) są w tym przypadku dwuargumentowe.

Operacja PG(S,n} odpowiada wykonaniu instrukcji:

- jeśli S > n, to S := S — n, t/; przeciwnym razie wstrzymaj działanie procesu wykonującego tę operację,

a operacja VG(S,n) odpowiada wykonaniu instrukcji

- jeśli są procesy wstrzymane w wyniku wykonania operacji PG(S,m), przy czym m < S + n, to wznów jeden z nich i S := S — m + n, w przeciwnym razie S := S + n.

Jednoczesne operacje semaforowe

Inne rozszerzenia idą w kierunku umożliwienia wykonania operacji opuszczenia jednocześnie na wielu semaforach.

Wykonanie operacji PAND(Sl,S2) (por. [PatiTl, MaOO87]) jest równoważne wykonaniu instrukcji

- jeśli S1 > 0 i S2 > O, to S1 := S1 — 1 ,S2 := S2 — 1 w przeciwnym razie wstrzymaj działanie procesu wykonującego tę operację.

Analogicznie wprowadza się operację POR(Sl,S'2), której wykonanie jest równoważne wykonaniu instrukcji (por. [Lipt74, Ma0087])

- jeśli Sl > O lub S2 > O, to odpowiednio albo Sl := Sl - l albo S2 := S2 — l w przeciwnym razie wstrzymaj działanie procesu wykonującego tę operację.

Zauważmy, że w przypadku, gdy oba semafory są podniesione, w wyniku wykonania operacji POR jeden z nich zostanie opuszczony, ale nie wiadomo który.

Uogólnienie operacji PAND i POR na wiele semaforów jest oczywiste. Jako parę dla operacji PAND wprowadza się czasem operację VAND, umożliwiającą jednoczesne podniesienie wielu semaforów. Jeszcze bardziej złożoną operację podnoszenia zaproponował Agerwala (por. [IsMa83]). Umożliwia ona jednoczesne podniesienie wskazanych semaforów pod warunkiem, że w tym samym czasie inne wskazane semafory są opuszczone. Tego typu operacje są dostępne w systemie Unix.

Od implementacji operacji P i V wymagamy, aby miała własność żywotności. Tego samego powinno się wymagać od operacji PAND i POR. Na przykładzie zadania 3.3.4 pokażemy, że zapewnienie żywotności operacji PAND nie jest proste. W systemie Unix jest możliwość wykonywania jednoczesnych operacji semaforowych, ale ich realizacja nie ma własności żywotności.

3.1.5 Ograniczenia

Mimo że semafor jest faktycznie zmienną całkowitą (lub logiczną), nie można na nim wykonywać żadnych innych operacji, a zwłaszcza nie można testować jego wartości ani wykonywać na nim działań arytmetycznych. Nie ma także możliwości stwierdzenia, ile procesów zostało wstrzymanych na danym semaforze.

Do synchronizacji procesów w rozwiązaniach niektórych klasycznych problemów, takich jak wzajemne wykluczanie, problem producenta i konsumenta czy problem pięciu filozofów, wystarczą same semafory. Jest jednak wiele problemów, w których do podjęcia właściwej decyzji jest potrzebna wartość semafora lub informacja o liczbie wstrzymanych procesów. Trzeba wówczas oprócz semafora wprowadzić zwykłe, pomocnicze zmienne całkowite, w których będą pamiętane odpowiednie wartości. Programista sam musi jednak zadbać o to, aby zmienne te były modyfikowane przy każdej operacji P lub V. Wszelkie operacje na tych zmiennych muszą być sekcją krytyczną procesu, co oznacza, że do ich ochrony trzeba użyć jeszcze jednego semafora. Program współbieżny staje się przez to bardziej skomplikowany i traci na przejrzystości.

W sytemie Unix jest możliwość zarówno odczytywania wartości semafora, jak i stwierdzania, czy są procesy czekające na jego podniesienie. Wszystkie zadania w rozdz. 8, w którym omawiamy semafory unixowe, rozwiązano za pomocą samych semaforów, bez użycia zmiennych pomocniczych do synchronizacji procesów.

3.2 Przykłady

3.2.1 Wzajemne wykluczanie

Semafor wymyślono po to, aby w prosty sposób można było za jego pomocą zrealizować wzajemne wykluczanie. Do wzajemnego wykluczania dowolnej liczby procesów wystarczy jeden semafor binarny. Rozwiązanie problemu wzajemnego wykluczania wygląda następująco:

```
const N = ?;       {liczba procesów}
var S: binary semaphore := 1;

process P(i:l..N);
begin
 while true do begin
 wlasne_sprawy;
 PB(S);
 sekcja_krytyczna;
 VB(S)
 end
end;
```

Ponieważ semafor S ma wartość początkową 1, tylko jeden proces będzie mógł wykonać operację PB(S) bez wstrzymywania, pozostałe będą musiały czekać, aż proces ten wykona operację VB(S).

3.2.2 Producenci i konsumenci

Kilka rozwiązań problemu producenta i konsumenta za pomocą semaforów można znaleźć w książce [BenA89]. Tu podamy rozwiązanie dla Inifora N-elementowego, N > 1. Semafor WOLNE będzie podniesiony, jeśli w buforze będą wolne elementy. Semafor PEŁNE będzie podniesiony wtedy, gdy w buforze będą zapełnione elementy. Aktualne wartości tych semaforów będą wskazywać na liczby odpowiednio wolnych i zajętych elementów bufora.

```
const N = ?;       {rozmiar bufora}
```

```
var WOLNE: semaphore := N;
 PEŁNE: semaphore := 0;
 bufor: array[l..N] of porcja;

process PRODUCENT;
var p: porcja;
 j: 1..N := 1;
begin
 while true do begin
 produkuj(p);
 P(WOLNE);
 bufor [j] := p;
 j := j mod N + l;
 V(PEŁNE)
 end
end;
process KONSUMENT;
var p: porcja;
 k: 1..N := 1;
begin
 while true do begin
 P(PEŁNE);
 p := bufor [k];
 k := k mod N + 1;
 V(WOLNE);
 konsumuj(p)
 end
end;
```

Przedstawione rozwiązanie jest poprawne tylko w przypadku jednego producenta i jednego konsumenta. Operacje na semaforach WOLNE i PEŁNE są tak wykonywane, że nigdy wartość zmiennej j w procesie PRODUCENT nie jest jednocześnie równa wartości zmiennej k w procesie KONSUMENT. Dopuszczamy tu więc równoczesne wstawianie do bufora i pobieranie z niego, bo operacje te odbywają się na różnych elementach bufora. Jeśli dopuszczamy wielu producentów i wielu konsumentów, zmienne wskazujące miejsca, z którego się wyjmuje i do którego się wkłada, muszą być zmiennymi globalnymi. Do ich ochrony jest potrzebny co najmniej jeden semafor. W podanym niżej rozwiązaniu użyliśmy dwóch semaforów, oddzielnego dla producentów i konsumentów.

```
const P = ?   {liczba producentów}
      K = ?   {liczba konsumentów}
      N = ?   {rozmiar bufora}
var WOLNE: semaphore:= N;
    PEŁNE: semaphore := 0;
 bufor: array[l..N] of porcja;
 j: integer := 1;
 k: integer := 1;
 CHROŃ_J: binary semaphore := 1;
 CHROŃ_K: binary semaphore := 1;

process PRODUCENT(i: 1..P);
var p: porcja;
begin
 while true do begin
 produkuj(p);
 P(WOLNE);
 PB(CHROŃ_J);
 bufor[j] := p;
 j := j mod N + 1;
```

```
  VB(CHROŃ_J);
  V(PEŁNE)
  end
 end;

 process KONSUMENT(i: 1..K);
 var p: porcja;
 begin
  while true do begin
  P(PEŁNE);
  PB(CHROŃ_K);
  p := bufor[k];
  k := k mod N + l;
  VB(CHROŃ_K);
  V(WOLNE);
  konsumuj(p)
  end
 end;
```

W tym rozwiązaniu także jest możliwe jednoczesne wstawianie i pobieranie, ale w każdej chwili tylko jeden PRODUCENT może wstawiać i tylko jeden KONSUMENT może pobierać. Wstawianie jest sekcją krytyczną producentów chronioną semaforem CHROŃ_J, a pobieranie jest sekcją krytyczną konsumentów chronioną semaforem CHROŃ_K.

3.2.3 Czytelnicy i pisarze

Rozwiązanie poprawne

Przedstawimy rozwiązanie zaproponowane przez Brinch Hansena [Brin79]. Ma ono tę interesującą własność, że z dokładnością do używanych zmiennych jest prawie identyczne dla czytelników i pisarzy. Jedyna różnica polega na tym, że pisarze są dodatkowo wstrzymywani na semaforze W, gwarantującym wyłączność pisania.

Każdy proces pragnący wejść do czytelni, stwierdziwszy, że nie ma oczekujących procesów z drugiej grupy, wpuszcza wszystkie oczekujące przed nim procesy tej samej grupy. Po wyjściu z czytelni proces, który stwierdzi, że jest ostatni ze swojej grupy, wpuszcza wszystkie oczekujące procesy drugiej grupy (oczywiście w przypadku pisarzy wchodzą oni do czytelni po kolei). W rezultacie, jeśli w czytelni są czytelnicy, to może do nich dołączyć nowy czytelnik tylko wtedy, gdy nie czekają pisarze. Jeśli pisarze czekają, to po wyjściu ostatniego czytelnika będą kolejno wpuszczeni do czytelni.

```
const M = ?;      {liczba czytelników}
      P = ?;      {liczba pisarzy}
var ac: integer := 0;   {aktywni czytelnicy}
    dc: integer := 0;   {działający czytelnicy}
    ap: integer := 0;   {aktywni pisarze}
    dp: integer := 0;   {działający pisarze}
 CZYT: semaphore := 0;  {wstrzymuje czytelników}
  PIS: semaphore := 0;  {wstrzymuje pisarzy}
 CHROŃ: binary semaphore := l;{do ochrony zmiennych}
     W: binary semaphore := 1; {do wykluczania pisarzy}

process CZYTELNIK (i:1..M) ;
begin
 while true do begin
 własne_sprawy;
```

```
 PB(CHROŃ);
  ac := ac + 1;
  if ap = O then
  while dc < ac do begin
   dc := dc + 1;   {wpuszczenie wszystkich}
   V(CZYT)         {czytelników przed sobą}
  end;             {i otwarcie sobie drogi}
 VB(CHROŃ);
 P(CZYT);
 czytanie;
 PB(CHROŃ);
  dc := dc - 1;
  ac := ac - l;
  if dc = O then
  while dp < ap do begin
   dp := dp + 1;   {wpuszczenie wszystkich}
   V(PIS)          {oczekujących pisarzy}
  end;
  VB(CHROŃ)
 end
end;

process PISARZ(i:1..P);
begin
 while true do begin
 własne.sprawy;
 PB(CHROŃ);
   ap : = ap + l;
   if ac = O then
   while dp < ap do begin  {tu mógłby być if,}
    dp := dp + 1;          {ale dla symetrii jest}
    V(PIS)                 {instrukcja while}
   end;
  VB(CHROŃ);
  P(PIS);
  PB(W);
  pisanie;
  VB(W);
  PB(CHROŃ);
   dp := dp - 1;
   ap := ap - 1;
   if dp = O then
   while dc < ac do begin
    dc := dc + 1;   {wpuszczenie wszystkich}
    V(CZYT)         {czekających czytelników}
   end;
  VB(CHROŃ)
  end
 end;
```

Jeżeli liczba czytelników M jest znana lub w czytelni może przebywać co najwyżej M czytelników i (w obu przypadkach) M jest nieduże, można podać znacznie prostsze rozwiązanie, w którym wystarczą dwa semafory. Semafor WOLNE wskazuje liczbę wolnych miejsc w czytelni. Semafor W służy do wzajemnego wykluczania pisarzy.

```
 const M = ?;     {liczba czytelników}
       P = ?;     {liczba pisarzy}
 var WOLNE: semaphore := M;  {liczba miejsc}
  W: binary semaphore := 1; {do wzajemnego wykluczania pisarzy}
```

```
process CZYTELNIK(i: 1..M);
begin
 while true do begin
 własne.sprawy;
 P(WOLNE);
 czytanie;
 V(WOLNE)
 end
end;
process PISARZ (i: 1..P);
var j: integer;
begin
 while true do begin
 własne_sprawy;
 PB(W);
 for j := l to M do P(WOLNE);
 pisanie;
 for j := l to M do V(WOLNE);
 VB(W)
 end
end;
```

Podniesienie semafora WOLNE oznacza zdobycie wolnego miejsca, a więc uprawnia czytelnika do wejścia do czytelni. Pisarz musi zająć wszystkie miejsca w czytelni. Robi to jednak stopniowo. Z własności żywotności implementacji semafora wynika, że pisarzowi uda się w końcu zająć wszystkie miejsca w czytelni. Z tej samej własności wynika, że każdy czytelnik kiedyś wejdzie do czytelni. Rozwiązanie to jednak ma tę wadę, że nie wykorzystuje w pełni możliwości równoległego czytania. Pisarz wychodzący z czytelni zwalnia wszystkie miejsca. Zajmują je czytelnicy i kolejny czekający pisarz. Nie ma jednak pewności, że wszystkim oczekującym czytelnikom uda się wejść do czytelni. Pisarz może być szybszy i zająć większość miejsc lub nawet wszystkie miejsca.

Warto zauważyć, że co prawda drugie z podanych tu rozwiązań znacznie krócej się zapisuje, ale jego realizacja może być dużo dłuższa, gdyż czas wykonywania operacji semaforowych jest dłuższy niż czas wykonywania operacji na zwykłych zmiennych.

3.2.4 Pięciu filozofów

Rozwiązanie z możliwością blokady

Semafor binarny WIDELEC [i] będzie reprezentować widelec o numerze i.
Semafor podniesiony będzie oznaczał leżący widelec.
Podniesienie widelca będzie zatem operacją PB na semaforze WIDELEC[i] , a opuszczenie operacją VB na tym semaforze.

```
var WIDELEC: array[0..4] of binary semaphore:= (1,1,1,1,1);

process FILOZOF(i:0..4);
begin
 while true do begin
 myślenie;
 PB(WIDELEC[i]);
 PB(WIDELEC[(i+l) mod 5]);
 jedzenie;
 VB(WIDELEC[i]);
 VB(WIDELEC[(i+l) mod 5])
```

```
  end
 end;
```

Rozwiązanie poprawne

Semafor LOKAJ o początkowej wartości 4 dopuszcza co najwyżej czterech filozofów do współzawodnictwa o widelce. Jak wiadomo, takie postępowanie
chroni przed blokadą i zagłodzeniem.

```
var WIDELEC: array[0..4] of binary semaphore:= (1,1,1,1,1);
 LOKAJ: semaphore := 4;

process FILOZOF(i:0..4);
begin
 while true do begin
 myślenie;
 P(LOKAJ);
 PB(WIDELEC [i]);
 PB(WIDELEC[(i+l) mod 5]);
 jedzenie;
 VB(WIDELEC [i]);
 VB (WIDELEC [(i-H) mod 5]);
 V(LOKAJ)
 end
end;
```

3.3 Zadania

3.3.1 Implementacja semafora ogólnego za pomocą binarnego

Zapisz operacje V i P na semaforze ogólnym o wartości początkowej N za pomocą operacji na semaforze binarnym.

3.3.2 Implementacja semafora dwustronnie ograniczonego

Zapisz operację PD opuszczania semafora dwustronnie ograniczonego oraz operację VD podnoszenia semafora dwustronnie ograniczonego, posługując się jedynie operacjami P i V na zwykłych semaforach. Maksymalna wartość, jaką może przyjąć semafor, wynosi N.

3.3.3 Implementacja semafora uogólnionego

Zapisz operacje PG(S, m) i VG(S, n) na, semaforze uogólnionym S za pomocą operacji na semaforze zwykłym.

3.3.4 Implementacja semafora typu AND

Zapisz za pomocą operacji na zwykłych semaforach operacje jednoczesnego opuszczania PAND i jednoczesnego podnoszenia VAND dwóch semaforów ogólnych. Ich wartości początkowe wynoszą odpowiednio N i M. Zakładamy, że na tych samych semaforach będą

również wykonywane pojedyncze operacje P i V. Wskazówka: te operacje trzeba także na nowo zaimplementować.

3.3.5 Implementacja semafora typu OR

Zapisz za pomocą operacji na zwykłych semaforach operację alternatywnego podnoszenia jednego z dwóch semaforów POR. Ich wartości początkowe wynoszą odpowiednio N i M. Zakładamy, że na tych samych semaforach będą również wykonywane pojedyncze operacje P i V. Wskazówka: te operacje trzeba także na nowo zaimplementować.

3.3.6 Dwa bufory

W systemie są dwa bufory cykliczne b1 i b2. Procesy PI,...,PN produkują cyklicznie, niezależnie od siebie, porcje i wstawiają je do bufora b1. Proces S pobiera po dwie porcje z b1 i przetwarza je na jedną porcję wstawianą do b2. Proces K czeka na całkowite wypełnienie b1 po czym konsumuje cały pełny bufor b2 na raz. Napisać treści procesów P i V dla S i K.

3.3.7 Linia produkcyjna

Taśma produkcyjna stanowi zamkniętą pętlę i może pomieścić M obrabianych elementów (bufor cykliczny M-elementowy). Wokół taśmy chodzi N - robotników (procesów). Zadaniem pierwszego z nich (procesu P(0)) jest nakładanie na taśmę nowych nieobrobionych elementów. Następny realizuje pierwszą fazę obróbki, kolejny drugą fazę itd. Ostatni (proces P(N — 1)) zdejmuje gotowe elementy z taśmy robiąc miejsce pierwszemu, który znów może nakładać na taśmę nowe elementy. Każda następna faza obróbki może rozpocząć się dopiero po zakończeniu poprzedniej. Zapisz treści procesów-robotników.

1. Czym jest ograniczona prędkość obróbki w tym systemie?
2. Jakie znaczenie ma wielkość taśmy produkcyjnej?

Komentarz: Tego typu przetwarzanie nazywa się przetwarzaniem potokowym. Będziemy o nim mówić, więcej w rozdz. 6, 10 i 11.

3.3.8 Przejazd przez wąski most

(Zadanie to pochodzi z książki [Brin79].) Na drodze dwukierunkowej północ-południe znajduje się wąski most, przez który w danej chwili mogą jechać samochody tylko w jednym kierunku. Zsynchronizuj przejazd samochodów jadących z południa i północy tak, aby nie było kolizji i żeby samochód z każdego kierunku mógł w końcu przejechać przez most (czyli żeby nie było 'zagłodzenia).

3.3.9 Gra w „łapki"

Dwa procesy P1 i P2 dzielące wspólną zmienną łapki grają cyklicznie w „łapki". Początkowo wartość zmiennej łapki wynosi 2. Każdą rundę proces rozpoczyna od odjęcia jedynki od zmiennej łapki (cofnięcie ręki). Jeśli po tej operacji proces stwierdzi, że łapki = 1 (trafiony przeciwnik), to zwiększa licznik zwycięstw, ustawia łapki na 2, w przeciwnym razie runda będzie nierozstrzygnięta. Procesy muszą zsynchronizować się przed każdą następną rundą. Aby gra była sprawiedliwa, każdy proces po wykonaniu operacji łapki := łapki - 1 musi dać

szansę dostępu do wspólnej zmiennej przeciwnikowi (który może z tej szansy nie skorzystać). Zapisz algorytmy procesów P1 i P2.

3.3.10 Obliczanie symbolu Newtona

Następujący tekst jest modyfikacją zadania, które pochodzi od Manny i Pnueliego a zostało zamieszczone w książce [BenA89].

Zsynchronizować parę procesów współdziałających przy obliczaniu wartości symbolu Newtona ('^) = n(n - 1)(n - 2)...(n - k + 1)/n!, k > 1. Proces P2 zmienia wartość zmiennej globalnej y (początkowo równej 1) mnożąc ją przez kolejne liczby naturalne 2,..., A;. Proces P1 zmienia wartość zmiennej globalnej x, (początkowo równej n) mnożąc ją kolejno przez liczby n - 1, n - 2, ...,n — k + 1. Jeśli stwierdzi, że kolejne mnożenie spowodowałoby nadmiar, dzieli x przez y i zmienia wartość y na l (w tym momencie y powinno dzielić całkowicie x). Na końcu proces P1 dzieli x przez y i otrzymuje szukany wynik.
Wskazówka: i dzieli całkowicie iloczyn n(n - 1)...(n - i + 1).

3.3.11 Lotniskowiec

Lotniskowiec ma pokład o pojemności N samolotów oraz pas startowy. Pas startowy jest konieczny do startowania i lądowania samolotów, a może z niego korzystać w danej chwili tylko jeden samolot. Gdy liczba samolotów na lotniskowcu jest mniejsza niż K (O < K < Ar), priorytet w dostępie do pasa startowego mają samoloty lądujące, w przeciwnym razie startujące.
Zapisz algorytm samolotu, który funkcjonuje według schematu: postój - start - lot - lądowanie itd. Samolotów może być więcej niż Ar, wówczas ich część jest zawsze w powietrzu.

3.4 Rozwiązania

3.4.1 Implementacja semafora ogólnego za pomocą binarnego

Oto rozwiązanie, które zostało zamieszczone w książce [ShawSO] na str. 90:

```
const N = ?;     {wartość początkowa semafora}
var DOSTĘP: binary semaphore := 1;
    CZEKAJ: binary semaphore := 0;
    wartość: integer := N;

procedure P;
begin
 PB(DOSTĘP);
 wartość := wartość - 1;
 if wartość < O then begin
 VB(DOSTĘP);
 PB(CZEKAJ)
 end else
 VB(DOSTĘP)
end;

procedure V;
begin
```

```
 PB(DOSTĘP);
 wartość := wartość + 1;
 if wartość <= O then VB(CZEKAJ);
 VB(DOSTĘP)
end;
```

Użyto tu dwóch semaforów binarnych. Semafor CZEKAJ służy do wstrzymywania procesów, DOSTĘP zaś do ochrony zmiennej wartość przechowującej wartość semafora ogólnego. Ujemna wartość zmiennej wartość informuje o liczbie procesów czekających na podniesienie semafora.

Rozwiązanie to wydaje się oczywiste, okazuje się jednak, że jest niepoprawne bez względu na przyjętą definicję operacji semaforowych. Pokazuje to następujący przykład.

Załóżmy, że wartość początkowa symulowanego semafora wynosi zero (N = 0) i że z tego semafora mają korzystać cztery procesy. Załóżmy dalej, że najpierw dwa z nich wykonują operację P, a zaraz potem dwa pozostałe wykonują operację V. Scenariusz działań jest następujący: najpierw pierwszy proces zmniejsza zmienną wartość o 1, stwierdza, że jest ona teraz ujemna, więc podnosi semafor DOSTĘP i zatrzymuje się przed wykonaniem operacji PB(CZEKAJ). Teraz drugi proces zmniejsza zmienną wartość o 1 i także zatrzymuje się przed wykonaniem operacji PB (CZEKAJ). W tej chwili wartość wynosi -2, semafor DOSTĘP jest podniesiony, a CZEKAJ - opuszczony. Rozpoczyna się wykonywanie operacji V. W wyniku wykonania pierwszej z nich wartość wzrośnie do -1, a semafor CZEKAJ podniesie się. Po wykonaniu drugiej wartość wzrośnie do O, a na semaforze CZEKAJ ponownie wykona się operacja VB. W rezultacie po dwukrotnym wykonaniu operacji V wartość semafora CZEKAJ będzie wynosić 1, a to umożliwia wznowienie tylko jednego z dwóch pierwszych procesów, które właśnie dopiero teraz mogą przystąpić do wykonywania operacji P(CZEKAJ).

Możliwość realizacji przedstawionego scenariusza (trzeba przyznać, że dość złośliwego) wynika z faktu, iż nie zakładamy nic o wzajemnych prędkościach wykonywania procesów (por. p. 2.3.3), a więc między wykonaniem operacji VB (DOSTĘP) i PB (CZEKAJ) może zdarzyć się dowolnie wiele. Warto zauważyć, że przy założeniu, iż podnoszenie podniesionego semafora binarnego jest błędem, powyższy scenariusz spowoduje awaryjne przerwanie programu. Bez tego założenia opisany błąd może być długo nie wykryty.

W przypadku definicji klasycznej jest możliwy także inny scenariusz doprowadzający do błędu. Nie trzeba zakładać, że dwa pierwsze procesy nie przystąpiły jeszcze do realizacji operacji PB (CZEKAJ). Nawet jeśli procesy te zostaną wstrzymane w trakcie wykonywania operacji PB (CZEKAJ), to wykonanie operacji VB zmieniającej wartość semafora CZEKAJ nie gwarantuje, że któryś z czekających procesów od razu zakończy wykonywanie operacji PB i ponownie zmieni wartość tego semafora. Można więc powiedzieć, że w przypadku definicji klasycznej opisane rozwiązanie jest złe aż z dwóch powodów.

Drugie wydanie wspomnianej książki [BiSh88] zawiera rozwiązanie poprawione.

```
procedure P;
begin
 PB(DOSTĘP);
 wartość := wartość - 1;
 if wartość < O then begin
 VB(DOSTĘP);
 PB(CZEKAJ)
 end;     {tu poprzednio było else}
 VB(DOSTĘP)
end;

procedure V;
begin
 PB(DOSTĘP);
```

```
 wartość := wartość + 1;
 if wartość <= 0 then VB(CZEKAJ)
 else     {tu poprzednio nie było else}
 VB(DOSTĘP)
end;
```

Tutaj druga operacja V nie będzie mogła się rozpocząć dopóty, dopóki jeden z czekających procesów nie wykona do końca operacji PB (CZEKAJ), pierwsza operacja V nie podniesie bowiem semafora DOSTĘP.

3.4.2 Implementacja semafora dwustronnie ograniczonego

Posługiwanie się semaforem dwustronnie ograniczonym przez procesy wykonujące na nim operacje PD i VD dokładnie odpowiada posługiwaniu się buforem cyklicznym N-elementowym przez procesy producentów wstawiających do bufora i procesy konsumentów wyjmujących z bufora. Do zrealizowania semafora dwustronnie ograniczonego wystarczą więc dwa semafory zwykłe S i T (będą one odpowiadać semaforom WOLNE i PEŁNE z przykładu 3.2.2).

```
const N = ?;    {ograniczenie górne}
      K = ?;    {wartość początkowa, K <= N}
var S: semaphore := K;
    T: semaphore := N - K;

procedure PD;
begin
 P(S);
 V(T)
end;
procedure VD;
begin
 P(T);
 V(S)
end;
```

3.4.3 Implementacja semafora uogólnionego

Semafor ogólny S będzie służył do wstrzymywania procesów, a zmienna l będzie pamiętała aktualną wartość semafora uogólnionego. Dostęp do zmiennej l musi być chroniony, zatem jest potrzebny także semafor binarny CHROŃ. Będzie on chronił dodatkowo zmiennej c stosowanej do pamiętania liczby procesów wstrzymanych na semaforze S.

Proces wykonujący operację PG sprawdza, czy aktualna wartość l semafora jest mniejsza niż n. Jeśli tak, to musi on być wstrzymany na semaforze S. Przedtem jednak trzeba zwiększyć coli zwolnić dostęp do zmiennych. Po wznowieniu proces znów sprawdza, czy wartość semafora jest mniejsza niż n i ewentualnie znów czeka. Gdy wartość zmiennej l jest nie mniejsza niż n, proces odejmuje od niej n.

Proces wykonujący operację VG zwiększa wartość zmiennej l o n, a następnie zwalnia procesy wstrzymane na semaforze S. Procesy te sprawdzają, czy nowa wartość l jest dla nich odpowiednia i ewentualnie ją zmniejszają.

```
const N = ?;              {wartość początkowa}
var S: semaphore := 0;    {do wstrzymywania proc.}
CHROŃ: binary semaphore := 1; {do ochrony zmiennych}
```

```
 l: integer := N;    {wartość semafora}
 c: integer := 0;    {ile czeka na S}

procedure PG(n:integer);
begin
 PB(CHROŃ);
 while l < n do begin
 c := c + 1;    {będzie czekać na S}
 VB(CHROŃ);     {opuszcza sekcję kryt.}
 P(S);          {czeka}
 PB(CHROŃ)      {wraca do sekcji kryt.}
 end;
 l := l - n;      {już może zmniejszyć}
 VB(CHROŃ)       {opuszcza sekcję kryt.}
end;

procedure VG(n: integer);
begin
 PB(CHROŃ);
 l := l + n;      {dodaje}
 while c > 0 do begin
 V(S);          {wpuszcza wszystkich}
 c := c - l      {czekających na S}
 end;
 VB(CHROŃ)
end;
```

W przedstawionym rozwiązaniu proces wykonujący VG podnosi semafor S tyle razy, ile wynosi wartość zmiennej c. (Zauważmy, że nie musi ona być równa liczbie procesów wstrzymanych na semaforze S, gdyż procesy wykonujące procedurę PG mogą dowolnie długo przechodzić od instrukcji c := c + 1 do instrukcji P(S). Z tego powodu użyto tu semafora ogólnego a nie binarnego.) Wszystkie zwolnione w ten sposób procesy będą następnie wstrzymane na semaforze CHROŃ, który w chwili wykonywania VB(S) jest opuszczony. Po wykonaniu VB(CHROŃ) w procedurze VG któryś z tych procesów będzie mógł sprawdzić swój warunek w pętli. Przypominamy, że z definicji semafora nie wynika jednoznacznie, który to będzie proces. Może się tak zdarzyć, że pewien proces, który wywołał PG z dużą wartością n nigdy nie będzie tym pierwszym, a więc nigdy nie będzie mógł skończyć swojej operacji PG. Warto podkreślić, iż fakt, że realizacja semafora jest żywotna, nie ma tu nic do rzeczy. Gwarantuje ona bowiem jedynie, że jeśli proces jest wstrzymany na semaforze, na którym dostatecznie dużo razy wykonuje się operację VB, to kiedyś jego operacja PB się zakończy. W tym jednak przypadku procesy za każdym razem są wstrzymywane na nowo na semaforze CHROŃ w nieznanej kolejności i każdy z nich w końcu zostaje wznowiony.

Zaproponowane rozwiązanie nie wyklucza zatem zagłodzenia procesów. Nic dziwnego, procesy czekające na odpowiednią wartość zmiennej l są tu wybierane „na chybił trafił". Aby uniknąć zagłodzenia, trzeba wprowadzić pewne uporządkowanie. Wśród procesów czekających na odpowiednią wartość zmiennej l wyróżnimy ten pierwszy i będziemy wstrzymywać go na oddzielnym semaforze PIERWSZY. Zmienna logiczna jest będzie mówiła, czy jakiś proces jest wstrzymany na tym semaforze, zmienna ile — jaka wartość zmiennej l jest temu procesowi potrzebna. Zmienna logiczna przechodzi będzie wskazywać, czy w danej chwili jakiś proces przechodzi spod semafora S pod semafor PIERWSZY. Pozostałe zmienne będą miały to samo znaczenie, co poprzednio. (Zauważmy, że w tym przypadku semafor S może być semaforem binarnym.)

```
const N = ?;       {wartość początkowa}
```

```
 var S: binary semaphore := 0;  {do wstrzymywania}
  CHROŃ: binary semaphore := 1; {do ochrony zmiennych}
  l: integer := N;     {wartość semafora}
  c: integer := 0;     {ile czeka na S}
  PIERWSZY: binary semaphore := 0;{ten pierwszy}
 jest: boolean := false;          {czy jest pierwszy}
 przechodzi: boolean := false;  {czy przechodzi z S do PIERWSZY}
 ile: integer := 0;             {ile chce pierwszy}

procedure PG(n:integer);
begin
 PB(CHROŃ);
 if jest or przechodzi then begin {już ktoś czeka}
 c := c + 1;      {będzie czekać na S}
 VB(CHROŃ);      {opuszcza sekcję}
 PB(S);          {czeka}
 PB(CHROŃ);      {wraca do sekcji kryt.}
 przechodzi := false
 end;
 if l < n then begin
 jest := true;
 ile := n;         {podaje, na ile czeka}
 VB(CHROŃ);        {opuszcza sekcję}
 PB(PIERWSZY);     {teraz jest pierwszy}
 PB(CHROŃ);        {wraca do sekcji kryt.}
 end else
 l := l - n;      {może zmniejszyć}
 if c > 0 then begin
 c := c - l;
 przechodzi := true;
 VB(S)          {ktoś z S przejdzie}
 end;           {na początek}
 VB(CHROŃ)      {opuszcza sekcję}
end;

procedure VG(n: integer);
begin
 PB(CHROŃ);
 l := l + n;      {dodaje}
 if jest and (l >= ile) then begin
 jest := false;
 l := l - ile;     {zmniejsza pierwszemu}
 VB(PIERWSZY)
 end;
 VB(CHROŃ)
end;
```

Proces wykonujący PG najpierw sprawdza, czy jakiś inny proces jest już wstrzymany. Jeśli tak, to zostaje sam wstrzymany na semaforze S. Po wznowieniu sprawdza, czy wartość zmiennej l jest odpowiednia. Jeśli nie, to zostaje wstrzymany na semaforze PIERWSZY po zaznaczeniu, na jaką wartość czeka. Kiedyś jakiś inny proces wykonujący VG wznowi ten proces odejmując za niego wartość ile i zmieniając jest na true.

Warto w tym miejscu wyjaśnić kilka rzeczy. Zmienna jest musi otrzymywać wartość f alse w procedurze VG, aby uniemożliwić ponowne wykonanie PB (PIERWSZY) przy kolejnym wywołaniu VG. Zmniejszanie l o wartość ile jest wykonywane także w procedurze VG. Jest to konieczne, gdyż istnieje niebezpieczeństwo, że proces zwolniony spod semafora PIERWSZY sam nie zdąży tego zrobić. Między operacjami PB (PIERWSZY) i PB (CHROŃ) może się

bowiem zdarzyć wiele, w szczególności jakiś inny proces może wywołać procedurę PG, stwierdzić, że jest fałszywe a zmienna l ma odpowiednią wartość, i zmniejszyć 1.

Ten sam problem dotyczy przejścia spod semafora S pod semafor CHROŃ. Dlatego wprowadzono zmienną logiczną przechodzi. Dzięki niej procesy rozpoczynające wykonywanie PG w czasie, gdy jakiś proces przechodzi spod semafora S pod semafor CHROŃ, są wstrzymywane na semaforze S.

Zauważmy na koniec, że wszystkie użyte tu semafory są semaforami binarnymi, przy czym semafory S i PIERWSZY służą tylko do wstrzymywania procesów, i nigdy nie przyjmują wartości dodatniej.

Jak widać implementacja semaforów uogólnionych za pomocą semaforów zwykłych nie jest prosta. W rzeczywistości semafory takie realizuje się za pomocą mechanizmów niskiego poziomu (rozdz. 8)

3.4.4 Implementacja semafora typu AND

W prezentowanym rozwiązaniu semafory S[0] i S[l] będą semaforami, na których należy wykonać operacje PAND i VAND. Rzeczywiste wartości semaforów będą pamiętane w zmiennych ls[0] i ls[l]. Procesy czekające na wykonanie PAND będą wstrzymywane na semaforze A. Liczba procesów czekających na wykonanie operacji PAND będzie pamiętana w zmiennej ca.

Ponieważ zwykle operacje P i V wpływają na wykonanie operacji PAND i VAND, trzeba je więc na nowo zaimplementować. Te nowe operacje nazwiemy odpowiednio PSINGLE i VSINGLE. Liczby procesów czekających na wykonanie operacji PSINGLE będą pamiętane w zmiennych es [0] i es [1]. Do ochrony wszystkich zmiennych pomocniczych służy semafor W.

Niebanalnym problemem jest tu uniknięcie zagłodzenia. Mogłoby ono wystąpić, gdybyśmy zawsze preferowali np. procesy wykonujące PAND. Dlatego w sytuacji, gdy występuje konflikt między żądaniami wykonania operacji pojedynczych i podwójnych, bierze się tu pod uwagę, jakie operacje były wykonywane ostatnio (tzn. jeśli ostatnio wykonywaną operacją na podnoszonym semaforze była operacja PSINGLE, to priorytet ma operacja PAND i odwrotnie). W tym celu dla każdego semafora S wprowadzono zmienną logiczną pój, która informuje, czy ostatnio wykonano na nim operację pojedynczą.

Trzeba jednak jeszcze zdecydować, co zrobić w sytuacji konfliktowej, jeśli równocześnie są procesy wstrzymane podczas wykonywania operacji PAND oraz PSINGLE na każdym z semaforów, ponadto na jednym z nich wykonano poprzednio operację pojedynczą, a na drugim podwójną. Wtedy zawsze wznawiamy procesy wykonujące operacje pojedyncze. Dzięki temu ostatnio wykonanymi operacjami staną się operacje pojedyncze i (po wykonaniu VSINGLE na każdym z semaforów) w końcu będzie mogła wykonać się operacja PAND. Gdybyśmy w takiej sytuacji wznowili proces czekający na wykonanie operacji podwójnej, to mogłoby dojść do zagłodzenia. Wyobraźmy sobie, że proces P, który wykonał PAND, podniesie najpierw tylko jeden semafor, np. S[0], wykonując VSINGLE. Jakiś inny proces wykonuje na tym semaforze PSINGLE (może to zrobić, gdyż na tym semaforze ostatnio wykonano operację podwójną). Jeżeli po zwolnieniu przez proces P semafora S[l] okaże się, że są procesy wstrzymane podczas wykonywania operacji PAND oraz PSINGLE na każdym z semaforów, to wznowimy proces czekający na wykonanie operacji podwójnej. W ten sposób procesy, które chcą korzystać tylko z semafora S[l] , zostaną zagłodzone.

Procedury YSINGLE i VAND tylko zwiększają wartości semaforów, a potem wywołują procedurę daj, w której następuje próba wznowienia czekających procesów (jeśli takie są). Próba ta nie zawsze musi zakończyć się powodzeniem. Jest tak wtedy, gdy priorytet ma operacja PAND, ale drugi semafor nie jest jeszcze podniesiony.

W przypadku semafora ogólnego do tego, by proces wykonał procedurę P bez wstrzymywania, wystarczy, aby wartość semafora była większa od zera. W procedurze PSINGLE sprawdza się dodatkowo, czy w ten sposób nie pozbawi się priorytetu procesów czekających na PAND. PSINGLE można zatem wykonać bez wstrzymywania, gdy dodatkowo wartość semafora przekracza liczbę procesów oczekujących na semaforze A (zmniejszając tę wartość na pewno nie przeszkodzi się procesom czekającym na PAND) lub gdy ostatnio wykonywaną operacją na danym semaforze była operacja PAND (wtedy PSINGLE ma priorytet).

```
 const M = ?;    {wartość początkowa S[0]}
       N = ?;    {wartość początkowa S[l]}
 type nrsem = 0..l;

 var A: semaphore := 0; {do wstrzymywania procesów
        wykonujących PAND>
 S: array[nrsem] of semaphore := (0,0);
       {oraz wykonujących PSINGLE}
 W: binary semaphore := 1;   {do wzajemnego wykluczania}
 Is: array[nrsem] of integer := (M,N);  {wartości semaforów}
 ca: integer := 0;  {liczba procesów czekających na oba semafory oraz}
 es: array[nrsem] of integer := (0,0);    {na pojedyncze semafory}
 pój: array[nrsem] of boolean := (true.true);
       {czy ostatnio wykonano PSINGLE}

procedure PSINGLE (i: nrsem);
begin
 PB(W);
 if (IsCi] > 0) and ((Is[i] > ca) or not pój [i]) then
 begin     {Is[i] przewyższa potrzeby czekających na PAND lub ostatnio
       na semaforze i nie było PSINGLE}
 Is[i] := Is[i] - 1;
 pój [i] := true;
 VB(W)
 end else begin    {trzeba czekać na S [i]}
 es [i] := es [i] + 1;
 VB(W);
 P(S[i])
 end
end;

procedure PAND;
begin
 PB(W);
 if ls[0] * ls[l] > O then begin
       {oba semafory podniesione}
 Is[O] := ls[0] - 1;
 ls[l] := Is[i] - 1;
 pój [O] := false;
 pój [l] := false;
 VB(W)
 end
 else begin    {trzeba czekać na A}
 ca := ca + 1;
 VB(W);
 P(A)
 end
end; {PAND}

procedure dajSINGLE (i: nrsem);
```

```
begin
 if ls[i] > O then begin
 es[i] := es [i] - 1;
 ls[i] := ls[i] - 1;
 pój [i] := true;
 V(S[i])
 end
end; {dajSINGLE}

procedure daj AND;
begin
 if ls[0] * ls[l] > O then begin
 ca := ca - 1;
 Is[O] := Is[O] - 1;
 ls[l] := ls[l] - 1;
 pój [0] := false;
 pój [1] := false;
 V(A)
 end
end; {daj AND}

procedure daj;
begin
 if ca = O then begin {nikt nie chce wykonać PAND}
 if cs[0] > O then dajSINGLE(O);
 if cs[l] > O then dajSINGLE(l)
 end
 else     {ktoś czeka na wykonanie PAND}
 if cs[0] * cs[l] > O then
       {są czekający na S[0] i S[l]}
 if pój [0] and pój [1] then
       {ostatnio na obu wykonano PSINGLE}
  daj AND    {można dać czekającemu na oba}
 else begin   {jeśli po j [0] and not poj[l]}
  dajSINGLE(O);  {lub na odwrót, to priorytet też}
  dajSINGLE(l)  {mają operacje PSINGLE}
  end
  else
  if cs[0] > O then  {ktoś czeka na S[0] , nikt na S[l]>
  if pój [0] then  {ostatnio na S[0] było PSINGLE}
  daj AND    {można dać czekającemu na oba}
  else     {ostatnio na S[0] wykonano PAND}
  dajSINGLE(O)
  else
  if cs[l] > O then  {ktoś czeka na S[l], nikt na S[0]}
  if pój [1] then  {ostatnio na S[l] było PSINGLE}
  daj AND    {można dać czekającemu na oba}
  else     {ostatnio na S[l] było PAND}
  dajSINGLE(l)
  else     {nikt nie czeka ani na S[0],}
  daj AND    {ani na S[2]}
 end; {daj}

 procedure YSINGLE (i: nrsem);
 begin
  PB(W);
  ls[i] := ls[i] + 1;
  daj;
  if (ls[i] > ca) and (cs[i] > 0) then dajSINGLE(i);
  VB(W)
 end;
```

```
procedure VAND;
begin
 PB(W);
 ls[0] := ls[0] + 1;
 ls[l] := ls[l] + 1;
 daj;
 VB(W)
end;
```

Semafory A i S służą praktycznie tylko do wstrzymywania procesów. Wydawać by się więc mogło, że wystarczyłyby tu semafory binarne. Ponieważ jednak procesy wykonujące procedurę PSINGLE mogą po operacji VB(W) zwlekać z wykonaniem operacji P (S [i]), a procesy wykonujące procedurę PAND mogą po wykonaniu operacji VB(W) zwlekać z wykonaniem operacji P (A), zatem semafory A i S muszą pamiętać liczbę wykonanych na nich operacji V odpowiednio w procedurach daj AND i daj SINGLE.

3.4.5 Implementacja semafora typu OR

Semafory S[0] i S[l] będą semaforami, na których należy wykonywać operacje POR. Rzeczywiste wartości semaforów będą pamiętane w zmiennych ls[0] i ls[l], które z kolei będą chronione przez semafor W. Ich wartości początkowe będą wynosić odpowiednio N i M. Ponieważ zwykle operacje P i V na tych semaforach wpływają na wykonanie operacji POR, dlatego trzeba je na nowo zaimplementować. Te nowe operacje nazwiemy odpowiednio PSINGLE i VSINGLE. Przyjmiemy tu dodatkowo, że procedura POR ma jeden parametr, na którym zwraca numer faktycznie opuszczonego semafora.

W celu zapobieżenia zagłodzeniu procesów, w rozwiązaniu użyto tablicy cobyło. Pamięta się w niej, czy w przypadku konfliktu podniesienie semafora S[i] spowodowało ostatnio zwolnienie procesu spod semafora LUB czy spod semafora S[i]. Należy bowiem w jakiś sposób ustalić, które z procesów, czy te czekające na PSINGLE, czy te czekające na POR, mają pierwszeństwo w chwili, gdy semafor się podniesie. Gdyby jednej z tych grup przyznać stały priorytet, druga mogłaby być zagłodzona. Dlatego w procedurze VSINGLE proces czekający na POR jest wznawiany tylko wtedy, gdy żaden proces nie czeka na wykonanie PSINGLE na podnoszonym semaforze, lub gdy poprzednio w podobnej sytuacji wybrano drugi semafor. Jeśli więc stale pod oboma semaforami czekają procesy na wykonanie PSINGLE oraz POR, to tylko w co drugim przypadku będzie zwalniany proces czekający na PSINGLE, a w co drugim czekający na POR.

Gdy oba semafory są podniesione, procedura POR powinna opuścić dowolny z nich. Aby nie był to zawsze ten sam semafor, wprowadzono zmienną który, wskazującą, od którego semafora należy zacząć sprawdzanie. Zmienia się ją za każdym razem, gdy któryś z semaforów był rzeczywiście opuszczony. Staramy się tu zatem o w miarę sprawiedliwe wykorzystanie obu semaforów. (Podobny rezultat można uzyskać wybierając semafor w sposób losowy z równym prawdopodobieństwem wylosowania każdego z nich.)

```
const M = ?;      {wartość początkowa S[0]}
      N = ?;      {wartość początkowa S[l]}
type nr.sem = 0..l;

var S: array [nr_sem] of semaphore := (0,0); {semafory właściwe}
 LUB: semaphore := 0;   {do wstrzymywania procesów wykonujących POR}
 Is: array[nr_sem] of integer := (M,N);
 {liczniki związane z S[i]}
```

```
  Hub : integer := 0;   {licznik związany z LUB}
  W: binary semaphore := 1; {do ochrony liczników}
 wybrał: integer := 0;   {wskazuje, który semafor podniesiono w VSINGLE}
 cobyło: array[nr_sem] of boolean := (true.true); {do pamiętania poprzednich
wyborów}
 który: nr_sem := 0;   {do zrównoważenia wykorzystania semaforów, gdy oba są
podniesione}
procedure POR(var i: nr_sem);
begin
 PB(W);
 if Is[który] > O then begin {podniesiony S[który]}
 Is [który] := Is [który] - 1; {będzie opuszczony}
 i := który;
 który := l - który;   {w przyszłości najpierw sprawdzimy drugi semafor}
 VB(W);
 end else
 if ls[l-który] > O then begin {podniesiony S[l-który]}
 ls[l-który] := ls[l-który] - 1;
 i := l - który;    {j.w.}
 który := i;
 VB(W);
 end else begin     {opuszczone oba}
 Hub : = Hub - 1;   {trzeba czekać}
 VB(W);
 P(LUB);
 i := wybrał;     {informacja od VSINGLE}
 VB(W)       {W był już opuszczony}
 end       {w YSINGLE}
end;

procedure PSINGLE(i: nr_sem);
begin
 PB(W);
 ls[i] := ls[i] - 1;
 if ls[i] >= O then VB(W)
 else begin      {S [i] był opuszczony}
 VB(W);
 P(S[i])      {trzeba czekać}
 end
end;
 procedure VSINGLE(i: nr_sem);
 begin
  PB(W);
  if (Hub < 0) and (ls[i] < 0) {konflikt - są procesy pod LUB i pod S[i]}
  then
  cobyło[i] := not cobyło[i]; {inne rozstrzygnięcie konfliktu niż poprzednio}
  if (Hub < 0) and ((ls[i] = 0) or not cobylo[i])
          {ktoś czeka na semaforze LUB oraz ma on tym razem pierwszeństwo lub nikt
          nie czeka na S[i]}
  then begin
  Hub := Hub + 1;
  wybrał := i;
  V(LUB)      {zostanie przepuszczony}
  end else
  if (ls[i] < 0) and ((Hub = 0) or cobyło[i])
          {ktoś czeka na semaforze S[i] oraz ma on tym razem pierwszeństwo lub nikt
          nie czeka na LUB}
  then begin
  ls[i] := ls[i] + 1;
  V(S[i]);      {zostanie przepuszczony}
  VB(W)
```

```
 end else begin     {nikt nie czeka}
 ls[i] := ls[i] + 1;
 VB(W)
 end
end;
```

Informację o tym, który semafor spowodował zwolnienie procesu spod semafora LUB przekazuje się w zmiennej wybrał. Musimy mieć tu gwarancję, że wartość tej zmiennej nie zostanie zmieniona do chwili odczytania jej przez proces czekający pod semaforem LUB w procedurze POR. W przedstawionym rozwiązaniu uzyskano to dzięki odpowiedniemu zwalnianiu dostępu do zmiennych globalnych chronionych przez semafor W. Proces wykonujący operację V(LUB) w procedurze VSINGLE nie podnosi semafora W, natomiast proces wykonujący operację P (LUB) w procedurze POR nie opuszcza go. Dzięki temu drugi z tych procesów „dziedziczy" po pierwszym dostęp do zmiennych globalnych, w tym także do zmiennej wybrał.

Semafory LUB i S służą tu praktycznie jedynie do wstrzymywania procesów, nie są tu jednak zadeklarowane jako semafory binarne z podobnych powodów jak semafory A i S w implementacji semafora typu AND.

3.4.6 Dwa bufory

Mamy tu do czynienia z dwoma układami producentów i konsumentów z różnymi zasadami konsumowania. Procesy P(i:l. .N) są producentami dla procesu S, który z kolei jest producentem dla procesu K. Pierwszy układ będziemy synchronizować za pomocą semaforów WOLNE1 i PEŁNE1. drugi za pomocą semaforów WOLNE2 i PELNE2. W tym drugim przypadku wystarczą semafory binarne. Proces S będzie sygnalizował procesowi K jedynie całkowite wypełnienie bufora b2. Z kolei proces K będzie sygnalizował procesowi S całkowite opróżnienie bufora b2. Zmienna n służy do liczenia zapełnionych porcji w buforze b2. Jeśli n = O, proces S musi czekać na to, aż proces K skonsumuje cały bufor b2.

Zakładamy tutaj istnienie procedur produkuj , przetwórz i konsumuj.

```
const W1 = ?;     {rozmiar bufora b1>
      W2 = ?;     {rozmiar bufora b2>

var bl: array[l..W1] of porcja;
 b2: array[l..W2] of porcja;
 WOLNE1: semaphore := W1;
 PELNE1: semaphore := 0;
 WOLNE2: binary semaphore := 1;
 PELNE2: binary semaphore := 0;
 j : integer := l;
 CHROŃ.J: binary semaphore := 1;

process P(i:l..N) ;
var pl: porcja;
begin
 while true do begin
 produkuj(pl);
 P(WOLNEl);     {czeka na wolne miejsce}
 PB(CHROŃ.J);
 bl[j] := pl;
 V(PELNE1);     {sygnalizuje zapełnienie}
 j := j mod W1 + 1;
 VB(CHROŃ_J)
```

```
 end
end;
process S;
var pl,p2,p3: porcja;
 m: 1..W1 := 1;
 n: 1..W2 := 0;
begin
 while true do begin
 P(PEŁNEl);
 P(PELNEl);     {czeka na 2 pełne elementy}
 pl := b1[m];
 m := m mod W1 + l;
 p2 := b1[m];
 m := m mod W1 + l;
 V(WOLNE1);     {sygnalizuje opróżnienie}
 V(WOLNE1);
 przetwórz(p1,p2,p3);
 if n = O then PB(WOLNE2); {czeka na wolny b2}
 n := n + 1;
 b2[n] := p3;
 if n = W2 then begin
  VB(PELNE2);    {sygnalizuje wypełnienie}
  n := O
 end
 end
end;

process K;
begin
 while true do begin
 PB(PELNE2);
 konsumuj(b2);
 VB(WOLNE2)
 end
end;
```

Zmienna j wskazująca miejsce, w które ma wkładać proces P, jest zmienną globalną chronioną przez semafor CHROŃ_J, gdyż korzystają z niej wszystkie procesy P. Zmienna m wskazująca miejsce, z którego wyjmuje proces S, jest lokalna w tym procesie.

3.4.7 Linia produkcyjna

Jest to problem podobny do problemu producenta i konsumenta. Każdy proces jest tu konsumentem dla poprzednika i producentem dla następnika.

Można sobie wyobrazić, że procesy „biegają" naokoło bufora cyklicznego w tym samym kierunku w kolejności wzrastania ich numerów, ale nie mogą się nawzajem przeganiać. Wartości semaforów określają dystans między procesami. Jeśli dystans zmaleje do zera, to proces o wyższym numerze musi być wstrzymany, by nie przegonić procesu o niższym numerze. Początkowo wszystkie procesy są wstrzymane przy pierwszej porcji bufora, a jedynie proces P(0) ma do pokonania odległość M, by dogonić proces P(N-1).

```
const N = ?;        {liczba procesów}
      M = ?;        {rozmiar bufora}
var bufor: array[0..M-l] of porcja;
 S: array[0..N-l] of semaphore := (M, (N-l)*0);
```

```
process P(i: 0..N-l) ;
var mojaporcja: O..M-1 := 0;
begin
 while true do begin
 P(S[i]);
 przetwórz(bufor[mojaporcja]);
 V(S[(i+l) mod N]);
 mojaporcja := (mojaporcja +1) mod M
 end
end;
```

ad. 1. Procesy nie mogą się przeganiać, więc prędkość obróbki jest ograniczona prędkością najwolniejszego z nich.

ad. 2. Zbyt mały bufor będzie powodować przestoje w pracy najwolniejszego procesu. Zwiększanie bufora ponad pewną wielkość, przy której najwolniejszy proces nie ma przestojów, nie zwiększy już szybkości przetwarzania w całym systemie.

3.4.8 Przejazd przez wąski most

Po dokładniejszym zanalizowaniu tego zagadnienia można stwierdzić, że mamy tu do czynienia z wariantem problemu czytelników i pisarzy, w którym dopuszcza się, aby pisarze pisali wspólnie. (Inaczej mówiąc mamy tu dwie grupy wykluczających się czytelników.) Rozwiązanie uzyskamy usuwając, w sprawiedliwym rozwiązaniu problemu czytelników i pisarzy z p. 3.2.3 semafor W i operacje na nim. Algorytm czytelnika będzie wówczas odpowiadał algorytmowi samochodu jadącego z północy, a algorytm pisarza — algorytmowi samochodu jadącego z południa.

3.4.9 Gra w „łapki"

Użyjemy tu trzech semaforów. Semafor DOSTĘP będzie służył do ochrony zmiennej łapki, semafor KONIEC do wstrzymywania procesu nr l do chwili, aż proces nr 2 skończy swoją rundę gry, semafor CZEKA będzie służył do wstrzymywania procesu nr 2, aż proces nr l ustawi z powrotem wartość zmiennej łapki na 2.

```
var łapki: integer := 2;
 DOSTĘP: binary semaphore := 1;
        {chroni dostępu do łapki}
 KONIEC: binary semaphore := 0;
        {sygnalizuje koniec P(2)}
 CZEKA: binary semaphore := 0;
        {wstrzymuje proces P(2)}

process P(i: l..2);
var licznik: integer := 0; {do liczenia punktów}
begin
 while true do begin
 PB(DOSTĘP);
 łapki := łapki - 1;  {zabranie ręki}
 VB(DOSTĘP);    {danie szansy przeciwnikowi}

 PB(DOSTĘP);
 if łapki = l then   {trafiony!}
  licznik := licznik + 1; {jest punkt}
```

```
 VB(DOSTĘP)

 case i of    {synchronizacja przed nową}
 1: begin     {rundą}
   PB(KONIEC);   {Pl czeka, aż P2 skończy}
   łapki := 2;   {ustawia się na nowo}
   VB(CZEKA)   {daje sygnał do nowej rundy}
  end;
 2: begin
   VB(KONIEC);   {P2 kończy}
   PB(CZEKA)   {P2 czeka na sygnał od Pl}
  end
 end
 end
end;
```

Każdy proces w danej rundzie najpierw odejmuje jedynkę od zmiennej łapki a potem sprawdza, czy jej wartość wynosi jeden. Jeśli oba procesy odejmą jedynkę przed sprawdzeniem, oba stwierdzą, że wartość, jest różna od jedności, a więc będzie remis. Jeśli jeden z nich zdąży sprawdzić wartość, zanim drugi odejmie jedynkę, stwierdzi, że wygrał i zwiększy licznik zwycięstw. Wówczas przeciwnik przed swoim sprawdzeniem musi sam odjąć jedynkę, a więc nie ma już szansy stwierdzić, że zmienna łapki ma wartość jeden.

Po sprawdzeniu zmiennej łapki proces l czeka, aż proces 2 także ją sprawdzi, po czym ustawia ją na nowo i daje sygnał do nowej rundy. Proces 2 informuje l, że już sprawdził, po czym czeka na sygnał do nowej rundy.

3.4.10 Obliczanie symbolu Newtona

Inicjatywę ma tu proces P1, musi więc wskazywać procesowi P2, jak daleko może on zajść w swoich obliczeniach. Do wstrzymywania i popychania procesu P2 służy semafor POPYCHACZ. Dostęp do wspólnej zmiennej y chroni semafor DOSTĘP. Semafor KONIEC jest używany do wstrzymywania procesu P1 do chwili, gdy proces P2 całkowicie zakończy swoje obliczenia. Semafor RÓŻNEOD1 powoduje wstrzymywanie procesu P1 do chwili, gdy zmienna y osiągnie wartość różną od l (podzielenie x przez l nic nie da procesowi P2).

```
var x,y: integer;
 POPYCHACZ: semaphore := 0;
 DOSTĘP: binary semaphore := 1;
 KONIEC: binary semaphore := 0;
 RÓŻNEOD1 : binary semaphore := 0;

proces Pl;
var i: integer;
begin
 x := n;
 for i := n - l downto n - k -t- l do begin
 while maxint div i < x do
 begin       -[nadmiar>
  PB(RÓŻNEODl);    {czeka, aż y <> 1}
  PB(DOSTĘP);     {czeka na dostęp do y>
  x := x div y;    {podzielić zawsze wolno,}
  y := 1;      {bo P2 nie przegania Pl}
  VB(DOSTĘP)
 end;
 x : = x * i;
 V(POPYCHACZ)     {P2 może pójść dalej}
```

```
 end;
 PB(KONIEC);      {trzeba czekać na P2>
 x := x div y
end;

proces P2;
var i: integer;
begin
 y := 1;
 for i := 2 to k do begin
 P(POPYCHACZ);    {czeka na pozwolenie od Pl}
 PB(DOSTĘP);
 if y = l then YB(RÓŻNEODl); {teraz będzie y <> 1}
 y := y * i;
 VB(DOSTĘP)
 end;
 VB(KONIEC)
end;
```

Jeśli już pierwsze mnożenie daje nadmiar, opisana tu metoda obliczania symbolu Newtona nie przyniesie sukcesu. Zauważmy, że jeśli maxint div (n-1) < n, to oba procesy zablokują się wzajemnie, gdyż P l będzie czekał na podniesienie semafora RÓŻNEOD1, co może zrobić tylko P2, a P2 będzie czekał na podniesienie semafora POPYCHACZ, co może zrobić tylko Pl. Bez żadnej szkody można jednak odwrócić pętlę w procesie P l i rozpoczynać obliczanie licznika od n-k+1 w kierunku n, a wtedy pierwszym mnożeniem będzie (n-k+l)*(n-k+2) zamiast n*(n-l). Jeśli i to mnożenie daje od razu nadmiar, trzeba użyć innej metody obliczania dwumianu Newtona.

3.4.11 Lotniskowiec

Występują tu dwie grupy procesów (samolotów) korzystających z jednego wspólnego zasobu (pasa startowego). Po skorzystaniu z zasobu proces zmienia swoją grupę. Priorytet grupy zależy od jej aktualnej liczebności. Zmienna wolny określa stan pasa startowego. Semafor PAS_DLA_START służy do wstrzymywania samolotów startujących a zmienna chce_start wskazuje ich liczbę. Semafor PAS_DLA_LĄD jest używany do wstrzymywania samolotów lądujących a zmienna chce_ląd wskazuje ich liczbę. Semafor DOSTĘP chroni wspólne zmienne. Zmienna na_lotniskowcu określa liczbą samolotów na lotniskowcu.

```
 const K = ?;     {liczba graniczna}
   N = ?;     {pojemność pokładu}
   M = ?;     {liczba wszystkich samolotów}
var na.lotniskowcu: integer := ?;
 chce_start:  integer := 0;
 chce.ląd:  integer := 0;
 wolny:   boolean := true;
 DOSTĘP:   binary semaphore := 1;
 PAS_DLA_START: binary semaphore := 0;
 PAS_DLA_LĄD:  binary semaphore := 0;

process SAMOLOTU: 1..M);
 procedurę zwolnij_pas;
 begin
 if na_lotniskowcu < K then begin
  if chce_ląd > O then begin
  chce_ląd := chce_ląd - 1;
  VB(PAS_DLA_LĄD)
```

```
   end else if chce_start > 0 then begin
   chce_start := chce_start - 1;
   VB(PAS_DLA_START)
   end else wolny := true
  end else begin {priorytet dla startujących}
   if chce_start > 0 then begin
   chce_start := chce_start - 1;
   VB(PAS_DLA_START)
   end else if (chce ląd > 0) and (na.lotniskowcu < N)
   then begin
   chce_ląd := chce_ląd - 1;
   VB(PAS_DLA_LĄD)
   end else wolny := true
  end
  end; {zwolnij_pas}

 begin {SAMOLOT}
  while true do begin
  postój;
  PB(DOSTĘP);
  if wolny then begin
   wolny := false; VB(DOSTĘP)
  end else begin
   chce_start := chce_start + 1;
   VB(DOSTĘP);
   PB(PAS_DLA_START)
  end;
  start;
  PB(DOSTĘP);
  na_lotniskowcu := na_lotniskowcu - 1;
  zwolnij_pas;
  VB(DOSTĘP);
  lot;
  PB(DOSTĘP);
  if wolny and (na.lotniskowcu < N) then begin
   wolny := false; VB(DOSTĘP);
  end else begin
   chce.ląd := chce.ląd + 1;
   YB(DOSTĘP); PB(PAS_DLA_LĄD)
  end;
  lądowanie;
  PB(DOSTĘP);
  na_lotniskowcu := na_lotniskowcu + 1;
  zwolnij pas;
  VB(DOSTĘP)
  end{while}
 end; -(samolot}
```

4 Monitory

4.1 Wprowadzenie

4.1.1 Pojęcie monitora

Koncepcja monitora powstała w wyniku poszukiwań strukturalnego mechanizmu do synchronizacji procesów. Pierwsze pomysły pochodzą od P. Brinch Hansena, który najpierw zaproponował regiony krytyczne i warunkowe regiony krytyczne, a następnie uogólnił je w kierunku monitorów [Brin79]. W tym samym czasie, niezależnie od Brinch Hansena, mechanizm monitorów zaproponował C. A. R. Hoare [Hoar74].

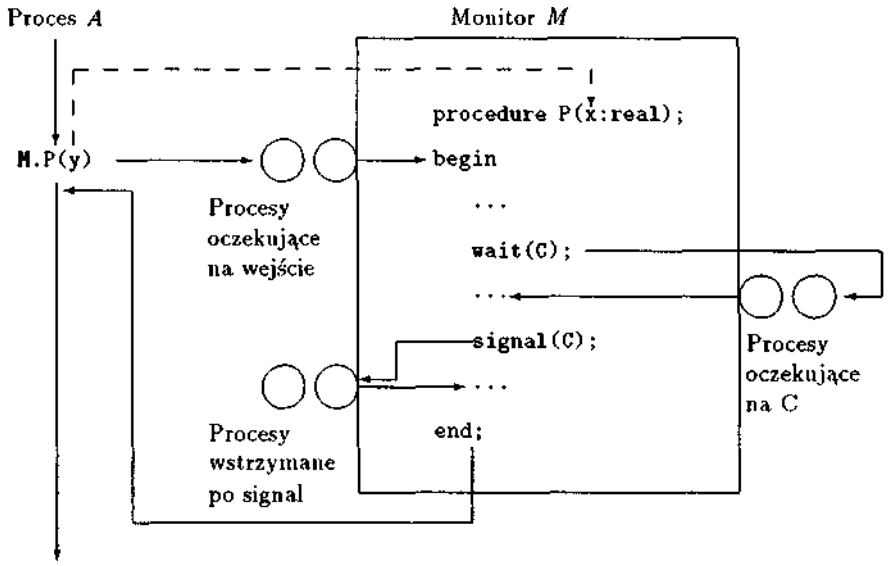
Monitor to zebrane w jednej konstrukcji programowej pewne wyróżnione zmienne oraz procedury i funkcje działające na tych zmiennych. Część tych procedur i funkcji jest udostępniana na zewnątrz monitora. Tylko ich wywołanie umożliwia procesom dostęp do zmiennych ukrytych wewnątrz monitora. Te cechy monitora upodabniają go do klas z języka Simula 67 lub do modułów z języka Modliła 2.

Cechą wyróżniającą jest fakt, iż wykonanie procedury monitora jest sekcją krytyczną wykonującego go procesu. Oznacza to, że w danej chwili tylko jeden spośród współbieżnie wykonujących się procesów może wykonywać procedurę monitora. Oprócz tego istnieje możliwość wstrzymywania i wznawiania procesów wewnątrz procedury monitorowej. Służą do tego zmienne specjalnego typu condition. Na zmiennych tego typu można wykonywać dwie operacje:

- wait(c) - powoduje wstrzymanie procesu wykonującego tę operację i wstawienie go na koniec kolejki związanej ze zmienną c, z jednoczesnym zwolnieniem monitora,
- signal(c) - powoduje wznowienie pierwszego procesu wstrzymanego w kolejce związanej ze zmienną c. Jeśli operacja ta nie jest ostatnią instrukcją procedury monitorowej, to proces wykonujący tę operację jest wstrzymany aż do chwili, gdy wznowiony przezeń proces zwolni monitor. Jeśli w chwili wywołania operacji signal żaden proces nie czeka w kolejce, to operacja ta ma efekt pusty.

Poza tym istnieje możliwość testowania kolejki, związanej ze zmienną typu condition. Uzyskuje się to przez wywołanie funkcji logicznej empty, która zwraca wartość true, gdy kolejka jest pusta, a wartość false w przeciwnym razie.

Mówiąc o monitorach często stosuje się potoczne sformułowania. I tak, będziemy mówić, że proces rozpoczynający wykonywanie procedury monitora „wchodzi do monitora", wykonujący procedurę monitora „przebywa wewnątrz monitora", a kończący wykonywanie procedury monitora „wychodzi z monitora".



RYS. 4.1. Przejście procesu przez monitor

Gdy jakiś proces P wykonuje w danej chwili procedurę monitora, jednocześnie mogą żądać dostępu do monitora inne procesy (rys. 4.1). Będą one czekać w trojakiego rodzaju kolejkach, z których jedna jest kolejką odwrotną (stosem). Procesy, które chcą rozpocząć wykonywanie procedury monitora, czekają w kolejce wejściowej, a te, które zostały wstrzymane w wynikli wykonania operacji wait, czekają w kolejkach związanych ze zmiennymi typu condition. Na wejście do monitora mogą czekać również procesy, które wykonały operację signal, ale same zostały wstrzymane. Zgodnie z definicją operacji signal są one odłożone na stos, muszą bowiem wracać do monitora w kolejności odwrotnej do tej, w jakiej go opuszczały. Jeśli proces P znajdujący się wewnątrz monitora wykona operację signal(c), będzie odłożony na ten stos procesów, a do monitora wejdzie pierwszy czekający w kolejce związanej ze zmienną c. Jeśli proces P opuści monitor albo wykonując operację wait(c) (będzie wówczas wstrzymany w kolejce związanej z c), albo kończąc wykonywanie procedury monitora, to do monitora wejdzie proces odłożony, który poprzednio wznowił proces P. Jeśli takiego procesu nie ma, oznacza to, że stos procesów odłożonych jest pusty i do monitora może wejść proces czekający w kolejce wejściowej

Można zapomnieć o dodatkowym stosie i wytłumaczyć to nieco krócej następująco. Procesy wykonujące operację final, która nie jest ostatnią instrukcją w procedurze monitora, są wstawiane na początek kolejki wejściowej do monitora. W chwili, gdy proces znajdujący się wewnątrz opuści monitor „na własne życzenie" (tzn. kończąc procedurę monitora lub wykonując operację wait), wchodzi do niego pierwszy proces z kolejki wejściowej.

Opisaliśmy monitor nie odwołując się do żadnego konkretnego języka programowania. Monitor jako strukturalne narzędzie do synchronizacji procesów zrealizowano w kilku językach. Poszczególne realizacje różnią się zwykle składnią i drobnymi szczegółami nie mającymi większego znaczenia dla sposobu posługiwania się tym mechanizmem. Dalej omówimy krótko sposoby deklarowania monitora w językach PascaLC, Concurrent Pascal, Pascal Plus, Modliła 2, Modula 3 oraz Concurrent Euclid. Jako przykład będziemy podawać deklarację monitora M zawierającego zmienną z typu integer i udostępniającego procedury Pl, P2, wewnątrz których są wykonywane operacje czekania i zwalniania na zmiennej W typu condition.

W przykładach i rozwiązaniach zadań skorzystamy z notacji zastosowanej w języku PascaLC. Robimy to z dwóch powodów. Po pierwsze, notacja ta jest zgodna z notacją użytą przez Ben-Ariego w jego książce [BenA89], którą gorąco polecamy. Po drugie, język PascaLC został zaimplementowany na maszynach typu IBM PC, a jego kompilator można uzyskać w

Instytucie Informatyki Uniwersytetu Warszawskiego. Co prawda z wymienionych tu języków najbardziej popularnym na komputerach typu IBM PC jest język Modula 2. Monitor nie jest w nim jednak wyróżnioną konstrukcją programową, a do tworzenia procesów korzystających z monitora stosuje się mechanizm niskopoziomowy. Te cechy powodują, że nie jest to dobry język do zapisywania przykładów zastosowania monitorów.

4.1.2 Ograniczenia

Każdej zmiennej typu condition odpowiada jedna kolejka, w której czekają wstrzymane procesy. Zwykle procesy są wstrzymywane, gdy nie jest spełniony jakiś warunek. Proces, który zmienia stan monitora, tak że warunek jest spełniony, wykonuje operację signal i wznawia, pierwszy ze wstrzymanych procesów. Są jednak przypadki, gdy okazuje się, że ten mechanizm jest zbyt słaby, by zwięźle zapisać rozwiązanie zadania,. Na przykład, gdy każdy z procesów oczekuje na inną wartość pewnej zmiennej, należałoby mieć dla każdej z tych wartości oddzielną kolejkę, co czasami może okazać się niewykonalne. W takiej sytuacji trzeba wszystkie procesy umieścić w jednej kolejce, ale jest potrzebny wówczas jakiś sposób stwierdzania, na co każdy z nich czeka.

Jest kilka rozwiązań tego problemu. Można z kolejką procesów związać dynamiczną strukturę, w której będą pamiętane potrzeby każdego z nich. Wtedy jednak programista musi wziąć na siebie cały wysiłek zorganizowania takiej dodatkowej struktury. Rozwiązaniem, które lepiej wykorzystuje możliwości monitora, jest podzielenie kolejki na dwie „podkolejki". W pierwszej będzie czekał tylko jeden proces, w drugiej pozostałe. Przechodząc z kolejki do kolejki, proces będzie w zmiennej pomocniczej pozostawiał informację, na jaki warunek czeka. W tym rozwiązaniu procesy są wznawiane w takiej samej kolejności, w jakiej były wstrzymywane. To podejście zastosowaliśmy w rozwiązaniach zadań 4.4.3 i 4.4.16.

Jeśli kolejność procesów czekających nie jest istotna, można na nie przerzucić obowiązek sprawdzenia stanu monitora. Proces zmieniający stan monitora będzie wówczas zawsze zwalniał proces czekający, a ten sam sprawdzi, czy warunek, na który czeka, jest już spełniony. Jeśli nie będzie spełniony, zwolni kolejny czekający proces, by też mógł sprawdzić warunek, a sam wykona znowu operację wait. W ten sposób wszystkie czekające procesy będą wznowione i wszystkie będą następnie wstrzymane (w odwrotnej kolejności niż były wznowione), z wyjątkiem być może jednego, który stwierdzi, że oczekiwany warunek jest już spełniony. Tego typu podejście zastosowaliśmy w rozwiązaniu zadania 4.4.11.

4.1.3 PascaLC

PascaLC [Kurp87] powstał w Instytucie Informatyki Uniwersytetu Warszawskiego. Jest to podzbiór języka Pascal, rozszerzony o instrukcję wykonania współbieżnego cobegin coend oraz o możliwość deklarowania monitorów i procesów w taki sam sposób, w jaki deklaruje się funkcje i procedury.

Deklaracje udostępnianych procedur poprzedza się słowem kluczowym export. Wewnątrz procesów do procedur monitorowych można odwoływać się kwalifikując nazwę procedury nazwą zawierającego ją monitora.

Nasz przykładowy monitor M zawierający zmienną z oraz udostępniający procedury Pl, P2, wewnątrz których są wykonywane operacje czekania i zwalniania na zmiennej W typu condition, ma w Pascalu.C następującą postać:

```
monitor M;
```

```
var z: integer;
 W: condition;
export procedure Pl;
begin
 ...wait(W); . . .
end; {Pl}
export procedure P2;
begin
 ...signal(W) ; . . .
end; {P2}
begin
 z := O
end; {M}
```

4.1.4 Concurrent Pascal

Concurrent Pascal zaproponowany przez Brincli Hansena [Brin75] jest jedną z pierwszych prób rozszerzenia Pascala o mechanizmy umożliwiające programowanie współbieżne. W jeżyku tym istnieje możliwość zdefiniowania specjalnego typu monitorowa/f i specjalnego typu procesowego na podobnej zasadzie jak definiuje się typ rekordowy. Nazwy udostępnianych procedur poprzedza się słowem kluczowym entry.

Typowi condition odpowiada w języku Concurrent Pascal typ queue. Na zmiennych tego typu można wykonywać operację delay (odpowiadającą operacji wait), operację continue (odpowiadającą operacji signal) oraz funkcję empty.

Istotnym ograniczeniem jest tu wymaganie, aby operacja continue była ostatnią wykonywaną instrukcją udostępnianej procedury lub funkcji. Kolejki w Concurrent Pascalu mogą być co najwyżej jednoelementowe. Jeśli chcemy wstrzymać w monitorze więcej niż jeden proces, należy zadeklarować w nim tablicę typu queue. Określenie sposobu szeregowania procesów pozostawia się programiście.

Typy procesowe i monitorowe mogą być sparametryzowane. Zwłaszcza typ procesowy może mieć parametr typu monitorowego, dzięki czemu można w definicji typu procesowego odwoływać się do procedur i funkcji wyjściowych monitora kwalifikując ich nazwy nazwą parametru.

Obiekty typu monitorowego i procesowego deklaruje się tak jak zwykłe zmienne, ale żeby móc z nich korzystać, trzeba je wcześniej zainicjować instrukcją init.

Nasz przykładowy monitor M ma w Concurrent Pascalu następującą postać:

```
type typm = monitor
 var z: integer;
  W: queue;
 procedure entry Pl;
 begin
 ...delay(W)...
 end;
 procedure entry P2;
 begin
 ...continue(W)
 end;
 begin
 z:=0
 end "typm";
var M: typm;
```

Aby skorzystać z tego monitora w procesie proc należy zdefiniować typ procesowy typp

```
type typp = process(mon:typm);

 mon.Pl;

  end "typp";
```

zadeklarować proces

```
var p:typp;
```

a następnie w programie głównym zainicjować monitor i proces

```
init M;
init proc(M);
```

Skrócony opis Concurrent Pascala można znaleźć w książce [IsMa83].

4.1.5 Pascal Plus

Pascal Plus jest kolejną próbą rozszerzenia języka Pascal o mechanizmy do programowania współbieżnego. Zrealizowano w nim idee, których autorem jest C.A.R. Hoare. Język ten nie ma niektórych ze wspomnianych ograniczeń Concurrent Pascala.

Podstawowym pojęciem jest koperta (envelope), czyli zbiór deklaracji zmiennych oraz procedur i funkcji zebranych razem w jednej konstrukcji programowej. Zmienne, procedury i funkcje, których nazwy poprzedzono gwiazdką są widoczne na zewnątrz koperty i można się do nich odwoływać kwalifikując je nazwą koperty. Koperta może mieć swoją własną treść, w której są inicjowane wartości zmiennych znajdujących się w jej wnętrzu.

Koperty są obiektami biernymi. Taką samą strukturę jak koperta ma proces. Treść procesu jest wykonywana współbieżnie z treścią innych procesów.

Monitor ma takie same własności jak koperta, z tą różnicą, że udostępniane przez niego procedury nie mogą być wykonywane jednocześnie przez kilka współbieżnych procesów.

Monitory (koperty, procesy) deklaruje się tak samo jak procedury. Jeśli po słowie kluczowym monitor (envelope, process) wystąpi słowo kluczowe module oznacza to, że jest to deklaracja samego obiektu. Jeśli brak tego słowa, mamy do czynienia z czymś w rodzaju deklaracji typu, a poszczególne obiekty trzeba utworzyć deklarując je po słowie kluczowym instance.

Omówiony typ condition jest w Pascalu Plus standardowym monitorem o nazwie Condition, który udostępnia następujące procedury:

t PWait(P:0. .Maxint) wstrzymuje proces z priorytetem P,
- Wait jest równoważne wykonaniu PWait(Maxint div 2),
- Signal odpowiada operacji signal,
- Empty odpowiada funkcji empty,
- Length zwraca długość kolejki,
- Priority zwraca priorytet pierwszego procesu w kolejce.

Jak widać, w Pascalu Plus jest możliwe wstrzymywanie procesów z priorytetem, co, jak się wkrótce przekonamy, znacznie upraszcza programowanie.

Nasz przykładowy monitor M ma w Pascalu Plus następującą postać:

```
monitor module M;
 var z: integer;
 instance W: Condition:
 procedure *P1;
 begin
```

```
... W.Wait...
end;
procedure *P2;
begin
... W.Signal...
end;
begin
z := O
end; {M}
```

W odróżnieniu od języka Concurrent Pascal instancje monitora są inicjowane automatycznie. Opis Pascala Plus można znaleźć w książce [BuEW88].

4.1.6 Modula 2

Zaprojektował ją Wirth jako rozszerzenie języka Pascal o pojęcie modułu, który w Moduli jest podstawową jednostką programową. Modni zawiera deklaracje typów, zmiennych, stałych i procedur oraz ciąg instrukcji, które są wykonywane w chwili wywołania procedury zawierającej dany moduł. Instrukcje te służą do nadawania wartości początkowych zmiennym lokalnym. Moduł może udostępniać na zewnątrz obiekty w nim zadeklarowane (zwłaszcza procedury) — muszą się one pojawić na liście EXPORT. Można w nim także odwoływać się do obiektów zadeklarowanych w innych modułach — muszą one wówczas być wymienione na liście IMPORT.

Standardowy moduł Processes umożliwia korzystanie z procesów współbieżnych. Udostępnia procedurę do tworzenia i uruchamiania nowego procesu oraz specjalny typ SIGNAL (który pełni rolę omówionego wcześniej typu condition) i operacje na nim. Dla zmiennej S typu SIGNAL procedura SEND(S) odpowiada operacji signal(S), procedura WAIT(S) — operacji w«z((S), funkcja logiczna Awaited(S) — funkcji empty(S). Dodatkowo wprowadzono procedurę Init(S) do inicjowania zmiennej S.

Jak widać pojęcie modułu jest bardzo zbliżone do omawianego już pojęcia monitora. Monitor uzyskuje się w Moduli 2 jako szczególny przypadek modułu wykorzystując możliwość nadawania modułom priorytetów. Moduł o najwyższym priorytecie ma bowiem tę własność, że wykonanie eksportowanej przez niego procedury nie może być przerwane przez żaden inny proces. Mamy więc w ten sposób spełniony warunek wzajemnego wykluczania się procedur monitorowych.

Nasz przykładowy monitor M ma w Moduli 2 następującą postać:

```
MODULE M[l];
 EKPORT P1.P2;
 IMPORT SIGNAL,SEND.WAIT,Init;
 VAR z: INTEGER; W: SIGNAL;
 PROCEDURĘ Pl;
 BEGIN
 ...
 WAIT(W)
 ...
 END;
 PROCEDURĘ P2;
 BEGIN
 ...
 SIGNAL(W)
 ...
 END;
BEGIN
```

```
  z := 0;
  Init(W)
END M;
```

W module, w którym chcemy skorzystać z tego monitora należy umieścić na liście IMPORT nazwy procedur Pl i P2.

Opis języka Modula 2 można znaleźć w książce [Wirt87].

4.1.7 Modula 3

Module 3 zaprojektowali L. Cardelli, J. Donahue, M. Jordan, B. Kalsow i G. Nelson w ramach wspólnego projektu ośrodków badawczych firm DEC i Olivetti. Punktem wyjścia były języki Modula 2 i Modula 2+ [RovnS6] oraz Mesa [LaR.e80]. Zrezygnowano tu z klasycznego pojęcia procesu na rzecz wątków (threads). Są to współbieżnie wykonywane obliczenia we wspólnej przestrzeni adresowej. Wątek ma zatem dostęp do zmiennych, które istniały w tej przestrzeni w chwili jego utworzenia, co powoduje, że wszelkie operacje na tych zmiennych wymagają synchronizacji.

Do zapewnienia wzajemnego wykluczania służą zmienne obiektowego typu MUTEX. Pełnią one rolę semaforów binarnych o wartości początkowej 1. Zmienna m typu MUTEX może wystąpić w instrukcji postaci LOCK m DO I END, która gwarantuje wzajemne wykluczanie wykonania ciągu instrukcji I i wykonania instrukcji zawartych w innych instrukcjach LOCK z tą samą zmienną m. Przed wykonaniem takiej instrukcji zmienna m jest blokowana, po wykonaniu — odblokowywana. Oprócz tego w Moduli 3 jest dostępny standardowy moduł, a dokładniej jego interfejs o nazwie Thread. Poza mechanizmami do tworzenia wątków udostępnia on m.in. typ obiektowy Condition oraz procedury:

- PROCEDURE Wait(m:MUTEX; c:Condition),która powoduje jednoczesne odblokowanie zmiennej m i wstrzymanie wątku w kolejce związanej ze zmienną c, a następnie ponowne zablokowanie m;
- PROCEDURE Signal(c:Condition), która powoduje wznowienie jednego lub więcej wątków czekających w kolejce związanej z c;
- PROCEDURE Broadcast(c:Condition), powodującą wznowienie wszystkich wątków oczekujących na c.

Wznowienie procesu nie oznacza tu jednak natychmiastowego jego wykonania. Wznowiony proces musi czekać na możliwość ponownego zablokowania zmiennej m. W tym czasie inne procesy mogą blokować tę zmienną i wykonywać swoje sekcje krytyczne. W praktyce wznowiony proces musi sam ponownie sprawdzić, czy zachodzi warunek, na który czekał. Autorzy języka pozostawiają swobodę implementacji procedury Signal, w szczególności może ona być identyczna z implementacją procedury Broadcast. Operacja Signal ma więc inne znaczenie niż w klasycznej definicji monitora. Należy także zauważyć, że brakuje tu odpowiednika funkcji empty.

W języku Modula 3 nie ma odrębnej konstrukcji językowej na wyróżnienie monitora. Efekt monitora uzyskuje się stosując odpowiednio zmienne typu MUTEX, instrukcje LOCK, obiekty typu Condition oraz wywołania procedur Wait, Signal i Broadcast. Nasz przykładowy monitor M można w Moduli 3 zrealizować następująco:

```
VAR M := NEW(MUTEX);
VAR z: INTEGER := 0;
VAR W := NEW(Thread.Condition);
```

```
PROCEDURE Pl() =
 BEGIN
 LOCK M DO
 ...Thread.Wait(m,W); .. .
 END;
 END Pl;

PROCEDURE P2() =
 BEGIN
 LOCK M DO
 ...Thread.Signal(W);...
 END;
 END P2;
```

O szczegółach Moduli 3 i zasadach programowania współbieżnego w tym języku można przeczytać w książce [NelsOl].

4.1.8 Concurrent Euclid

Concurrent Euclid został opracowany przez J. R. Cordy'ego i R. C. Holtaz Uniwersytetu w Toronto jako rozszerzenie języka Euclid [Lamp77] o mechanizmy umożliwiające programowanie współbieżne. W nim właśnie napisano jądro systemu Tunis, który jest zgodny z systemem operacyjnym Unix. Opis języka Concurrent Euclid oraz systemu Tunis zawiera książka [Holt83].

W Concurrent Euclid monitor definiuje się jako typ, podobnie jak w Concurrent Pascalu. Nazwy procedur,i funkcji widocznych na zewnątrz są wymieniane na liście exportowej w nagłówku bloku monitora. Dopuszcza się w nim, aby między procesem aktywowanym przez operację signal a procesem wywołującym tę operację do monitora wchodziły inne procesy. Nie ma też gwarancji, że procesy są wstrzymywane na zmiennej typu condition w kolejności zgodnej z kolejnością wywołań. Możliwe jest także wstrzymywanie procesów w kolejce priorytetowej przez wywołanie operacji wait z dwoma parametrami, przy czym drugi parametr jest wtedy liczbą naturalną określającą priorytet procesu, podobnie jak się to robi w Pascalu Plus.

Przykładowy monitor M ma w Concurrent Euclid następującą postać:

```
var M: monitor
  exports (P1.P2)
  var z: integer
  var W: condition
  procedure Pl =
   imports (var z, var W)
   begin ...wait(W)... end Pl
  procedure P2 =
   imports (var z, var W)
   begin... signal(W)... end P2
  initially
   imports (var z)
   begin z := O end
  end {M} monitor
```

4.1.9 Zestawienie

W tablicy 4.1 podajemy podstawowe cechy składni monitora w omówionych tu językach programowania: postać deklaracji monitora, sposób udostępniania procedur i funkcji, nazwę typu kolejkowego, nazwy operacji zawieszających i wznawiających procesy oraz nazwę funkcji testującej kolejkę.

TABLICA 4.1. Porównanie składni monitora w różnych jeżykach programowania

Język	Postać	Uzewnętrznienie	Kolejka	Zaw.	Wznów.	Test
PascaLC	monitor	export procedure	condition	wait	signal	empty
C. Pascal	typ	procedure entry	queue	delay	continue	empty
Pascal +	monitor	procedure *	Condition	.Wait	.Signal	.Empty
Modula 2	moduł	lista EXPORT	SIGNAL	WAIT	SEND	Awaited
Modula 3	typ	lista exports	Condition	Wait	Signal	empty
C. Euclid			condition	wait	signal	

4.2 Przykłady

4.2.1 Wzajemne wykluczanie

Wzajemne wykluczanie jest realizowane w monitorze w naturalny sposób, wynikający wprost z definicji monitora. Proces P, który chce korzystać z zasobu dzielonego wywołuje procedurę monitora, która realizuje dostęp do zasobu. Jeżeli żaden proces w tym czasie nie wykonuje procedury monitora, to rozpoczyna się realizacja procedury wywoływanej przez proces P. W przeciwnym razie proces P oczekuje w kolejce.

```
monitor WYKLUCZANIE;
var zasób: typ.zasobu;
export procedure DOSTĘP;
begin
 {działania na zasobie}
end; {DOSTĘP}
begin
end; {WYKLUCZANIE}

process P;
begin
 while true do begin
 własne_sprawy;
 WYKLUCZANIE.DOSTĘP
 end
end; {P}
```

Rozwiązanie problemu wzajemnego wykluczania jest jednak tak proste tylko wtedy, gdy z zasobu dzielonego może w danej chwili korzystać tylko jeden proces. Zasób (reprezentowany przez zmienne) może być wtedy ukryty w monitorze. Jest jednak wiele zagadnień, w których dostęp do zasobu dzielonego odbywa się według bardziej złożonych zasad. Większość prezentowanych tu problemów i zadań dotyczy właśnie takich sytuacji.

Implementacja semafora ogólnego

Monitor symulujący semafor musi dostarczać dwie procedury, odpowiadające operacjom P oraz V. Wewnątrz monitora jest zadeklarowana zmienna całkowita semafor,

służąca do przechowywania wartości semafora, i zmienna KOLEJKA typu condition, do ewentualnego wstrzymywania procesów wykonujących operację P. Po zwiększeniu wartości zmiennej semafor w procedurze V następuje bezwarunkowe wykonanie operacji signal(KOLEJKA). Jeżeli nie czeka żaden proces, to efekt tej operacji jest pusty.

Implementacja semafora za pomocą monitora różni się nieco od klasycznego semafora. Procesy są wznawiane w monitorze w takiej samej kolejności, w jakiej były wstrzymane. Natomiast w definicji semafora zakłada się tylko, że wstrzymany proces będzie wznowiony w skończonym czasie, czyli że nie będzie zagłodzony. W niektórych definicjach semafora mówi się o „sprawiedliwym" wznawianiu procesów. W monitorze wznawianie zrealizowano w najbardziej sprawiedliwej postaci.

```
monitor SEMAFOR;
var semafor: integer;
 KOLEJKA: condition;

export procedure P;
begin
if semafor = O then wait(KOLEJKA);
semafor := semafor - l
end; {P}

export procedure V;
begin
semafor := semafor + 1;
signal(KOLEJKA)
end; {V>

begin
semafor := O
end; {SEMAFOR}
```

4.2.2 Producenci i konsumenci

Bufor służący do komunikacji między producentami i konsumentami jest ukryty wewnątrz monitora BUFOR, który udostępnia dwie procedury: WSTAW i POBIERZ. Zmienna PRODUCENCI służy do wstrzymywania producentów, gdy bufor jest pełny, a zmienna KONSUMENCI - do wstrzymywania konsumentów, gdy bufor jest pusty. Zmienna ile wskazuje liczbę zajętych elementów bufora.

```
monitor BUFOR;
const N = ?;
var bufor: array[0..N-l] of porcja;
 ile, do.włożenia, do.wyjęcia: integer;
 PRODUCENCI, KONSUMENCI: condition;

export procedure WSTAW(element: porcja);
begin
if ile = N then wait(PRODUCENCI);
do_włożenia := (do.wlożenia + 1) mód N;
bufor[do_włożenia] := element;
ile := ile + 1;
signal(KONSUMENCI)
end; {WSTAW}
export procedure POBIERZ(var element: porcja);
begin
 if ile = O then wait(KONSUMENCI);
 do.wyjęcia := (do.wyjecia + 1) mód N;
```

```
 element := bufor [do_wyjecia];
 ile := ile - 1;
 signal(PRODUCENCI)
end; {POBIERZ}

begin
 ile := 0;
 do_włożenia := 0;
 do_wyjecia := O
end; {BUFOR}
```

4.2.3 Czytelnicy i pisarze

Czytelnia jest zasobem chronionym przez monitor CZYTELNIA, który udostępnia cztery procedury wywoływane odpowiednio przed rozpoczęciem czytania, po zakończeniu czytania, przed rozpoczęciem pisania i po zakończeniu pisania. Zmienne CZYTELNICY i PISARZE służą do wstrzymywania odpowiednio czytelników i pisarzy. Zmiennych czyta i pisze używa się do liczenia czytelników i pisarzy w czytelni.

Rozwiązanie z możliwością zagłodzenia pisarzy

```
monitor CZYTELNIA;
var czyta, pisze: integer;
 CZYTELNICY, PISARZE: condition;

export procedure POCZ_CZYTANIA;
begin
 if pisze > O then
 wait(CZYTELNICY);
 czyta := czyta + l
end; {POCZ_CZYTANIA}

export procedure KON_CZYTANIA;
begin
 czyta := czyta - 1;
 if czyta = O then
 signal(PISARZE)
end; {KON_CZYTANIA}
export procedure POCZ_PISANIA;
begin
if czyta+pisze > O then wait(PISARZE);
pisze := l
end; {POCZ_PISANIA}

export procedure KON_PISANIA;
begin
pisze := 0;
if empty(CZYTELNICY) then
 signal(PISARZE)
else
 repeat
 signal(CZYTELNICY)
 until empty(CZYTELNICY)
end; {KON_PISANIA}
begin
czyta := 0;
pisze := O
end; {CZYTELNIA}
```

Jeżeli w czytelni nie ma pisarza, to zgłaszający się czytelnik uzyskuje natychmiast dostęp do czytelni. Nie ma przy tym znaczenia, czy jakiś pisarz czeka w tym czasie, czy też nie. Jeśli po zakończeniu pisania pisarz stwierdzi, że jacyś czytelnicy czekają na dostęp do czytelni, to natychmiast wznawia ich wykonując w pętli repeat odpowiednią liczbę razy operację signal (CZYTELNICY). Każdorazowe wykonanie tej operacji powoduje, że pisarz zwalnia monitor, udostępniając go wznowionemu czytelnikowi. Czytelnik kończy wykonywanie procedury POCZ_CZYTANIA, a następnie opuszcza monitor, który znów zajmuje pisarz wykonujący pętlę repeat. W tym rozwiązaniu pisarz nie może zakończyć procedury KON_PISANIA, zanim nie wpuści do czytelni wszystkich czekających czytelników. Jeśli z procedury tej usuniemy pętlę repeat pozostawiając tylko jedną instrukcję signal (CZYTELNICY), a na końcu procedury POCZ.CZYTANIA dodamy instrukcję signal (CZYTELNICY), to otrzymamy rozwiązanie, w którym pisarz wychodzący z czytelni wpuszcza tylko pierwszego czekającego czytelnika, który z kolei wpuszcza następnego itd. Inaczej mówiąc, pisarz, zamiast sam wysyłać wszystkie sygnały, uruchamia ich „kaskadę".

Rozwiązanie poprawne

Aby uniknąć zgłodzenia trzeba podczas żądania dostępu do czytelni przez czytelnika sprawdzać, czy jakiś pisarz nie czeka już na wejście do czytelni. Jeśli tak, to czytelnik musi być wstrzymany. Natomiast po zakończeniu pisania są wznawiani przede wszystkim czytelnicy, podobnie jak w pierwszym rozwiązaniu. Wznawianie pisarza, gdy są czekający czytelnicy, mogłoby spowodować zagłodzenie czytelników. Rozwiązanie poprawne różni się zatem od rozwiązania poprzedniego tylko procedurą POCZ_CZYTANIA:

```
export procedure POCZ.CZYTANIA;
begin
 if not empty(PISARZE) or (pisze > 0) then
 wait CZYTELNICY);
 czyta := czyta + l
end; {POCZ.CZYTANIA}
```

4.2.4 Pięciu filozofów

Rozwiązanie z możliwością zagłodzenia

Dla każdego filozofa pamiętamy, ile leżących przy nim widelców jest wolnych - służy do tego tablica wolne. Proces, zgłaszając żądanie przydziału widelców, podaje swój numer. Na jego podstawie można stwierdzić, czy oba widelce są wolne. Jeśli nie, to proces jest wstrzymywany. Do wstrzymywania każdego procesu jest stosowana, odrębna zmienna. - element tablicy FILOZOF. Dzięki temu po odłożeniu widelców przez jednego z filozofów można próbować wznowić tylko jego sąsiadów.

```
monitor WIDELCE;
var wolne: array[0..4] of integer;
 FILOZOF: array[0..4] of condition;
 i: integer;

export procedure BIORĘ(i: integer);
begin
 while wolne [i] < 2 do wait(FILOZOF[i]);
 wolne [(i + 4) mod 5] := wolne [(i + 4) mod 5] - l;
 wolne [(i +1) mod 5] := wolne [(i +1) mod 5] – l;
```

```
end; {BIORĘ}

export procedure ODKLADAM(i: integer);
begin
 wolne [(i + 4) mod 5] := wolne[(i + 4) mod 5] + l;
 wolne [(i + 1) mod 5] := wolne[(i + 1) mod 5] + l;
 signal(FILOZOF[(i + 4) mod 5]);
 signal(FILOZOF[(i + 1) mod 5])
end; {ODKŁADAM}
begin
for i := O to 4 do wolne [i] := 2
end; {WIDELCE}
```

W tym rozwiązaniu filozof odkładający widelce wznawia bezwarunkowo swoich sąsiadów. Wznowiony filozof sprawdza, czy oba widelce są wolne i, jeżeli nie, znowu zostaje wstrzymany. Takie postępowanie może być kosztowne. Efektywniejsze jest sprawdzanie przez filozofa odkładającego widelce, czy jego sąsiedzi po wznowieniu będą mogli wziąć widelce, i wznawianie ich tylko w takim przypadku.

```
monitor WIDELCE;
var wolne: array[0..4] of integer;
 FILOZOF: array[0..4] of condition;
 i: integer;

export procedure BIORĘ(i: integer);
begin
 if wolne[i] < 2 then wait(FILOZOF[i]);
wolne[(i + 4) mod 5] := wolne[(i + 4) mod 5] - l;
wolne[(i + 1) mod 5] := wolne[(i +1) mod 5] - l
end; {BIORĘ}

export procedure ODKLADAM(i: integer);
begin
wolne[(i + 4) mod 5] := wolne[(i + 4) mod 5] + 1;
wolne[(i + 1) mod 5] := wolne [(i + 1) mod 5] + l;
 if wolne[(i+4) mod 5] = 2 then
 signal(FILOZOF[(i + 4) mod 5]);
 if wolne [(i + 1) mod 5] = 2 then
 signal(FILOZOF[(i + 1) mod 5])
end; {ODKŁADAM}
begin
for i := O to 4 do wolne [i] := 2
end; {WIDELCE}
```

Rozwiązanie poprawne

Warto zwrócić uwagę na znacznie większą złożoność tego rozwiązania w porównaniu z rozwiązaniem, w którym zastosowano semafory. Każdemu semaforowi odpowiada tu jeden warunek i jedna zmienna (logiczna lub całkowita).

```
monitor WIDELCE;
var WIDELEC: array[0..4] of condition;
 zajęty: array[0..4] of boolean;
 LOKAJ: condition;
 jest: integer;
 i: integer;

export procedure BIORĘ(i: integer);
```

```
begin
 if jest = 4 then wait(LOKAJ);
 jest := jest + 1;
 if zajęty [i] then wait(WIDELEC[i]);
 zajęty[i] := true;
 if zajęty[(i + 1) mod 5] then
 wait(WIDELEC[(i + 1) mod 5]);
 zajęty [(i +1) mód 5] := true
end; {BIORĘ}

export procedure ODKŁADAM(i: integer);
begin
 zajętyCi] := false;
 signal(WIDELEC[i]);
 zajęty [(i + 1) mod 5] := false;
 signal(WIDELEC[(i + 1) mod 5]);
 jest := jest - 1;
 signal(LOKAJ)
end; {ODKŁADAM}
begin
 for i := O to 4 do zajęty [i] := false;
 jest := O
end; {WIDELCE}
```

4.3 Zadania

4.3.1 Trzy grupy procesów

W systemie jest M zasobów typu A oraz N zasobów typu B. Procesy wykonywane w tym systemie należą do jednej z trzech grup:

- procesy żądające jednego zasobu typu A,
- procesy żądające jednego zasobu typu B,
- procesy żądające równocześnie jednego zasobu typu A i jednego zasobu typu B.

Korzystanie z zasobu wymaga wyłącznego dostępu do tego zasobu. Napisz monitor zarządzający dostępem do zasobów.

4.3.2 Przetwarzanie potokowe

N procesów przetwarza współbieżnie dane znajdujące się we wspólnym M-elementowym buforze B. Proces P(0) wstawia do elementu bufora dane, które mają przetworzyć kolejno procesy P(1), P(2), ..., P(N-2). Proces P(N-1) opróżnia element bufora robiąc miejsce procesowi P(0). Procesy P(i), i=0,. . . ,N-1, mają następującą treść:

```
const M = ?;
  N = ?;
process P(i: 0..N-l);
var j: integer;
begin
 j := i;
 while true do begin
 DAJWEŹ.WEŹ(i);
 przetwórz(j);
 DAJWEŹ.DAJ(i);
```

```
  j := j mod M + l
  end
 end;
```

Napisz monitor DAJWEŹ.

4.3.3 Producenci i konsumenci z losową wielkością wstawianych i pobieranych porcji

Napisz monitor synchronizujący działanie P producentów i K konsumentów korzystających ze wspólnego bufora, w którym może się pomieścić 2*M nierozróżnialnych elementów (ich kolejność nie jest istotna). Bufor ma być ukryty w monitorze. Producent wstawia do bufora losową liczbę elementów, konsument pobiera również losową liczbę elementów, przy czym w obu przypadkach liczby te są nie większe niż M. Wyjaśnij, po co to założenie. Można przyjąć, że istnieją procedury kopiujące do/z bufora (procedury te nie modyfikują liczby elementów zajętych w buforze). Parametr ile określa rzeczywistą liczbę elementów przenoszonych między buforem a porcją p:

```
type porcja = array [1..M] of element;
procedure do_bufora(ile:integer; p:porcja);
procedure z_bufora ile:integer; var p:porcja);
```

4.3.4 Producenci i konsumenci z asynchronicznym wstawianiem i pobieraniem

Wersja 1. Napisz monitor synchronizujący pracę P producentów i jednego konsumenta. Każdy producent cyklicznie wytwarza porcję informacji i umieszcza ją. we wspólnym buforze o pojemności N porcji. Jeżeli bufor jest pełny, a producent chce wstawić do niego porcję informacji, to jest ona tracona. Konsument cyklicznie pobiera porcję informacji i przetwarza ją. Zarówno wstawianie, jak i pobieranie z bufora trwają tak długo, że muszą, odbywać się poza monitorem. Konsument pobiera porcje informacji w takiej kolejności, w jakiej zostały wytworzone przez producentów.

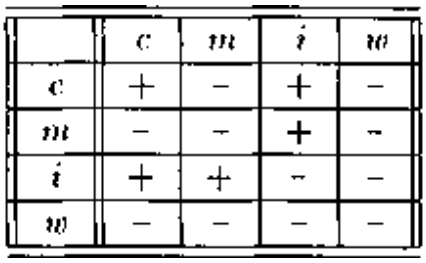
Wersja 2. Zadanie jak wyżej, ale nie ma tracenia porcji i jest K konsumentów. Porcje są przydzielane zgłaszającym się konsumentom w takiej kolejności, w jakiej rozpoczęło się ich wkładanie do bufora (tzn. jeśli producent P (5) zażądał miejsca na porcję w chwili 11, a producent P (2) w chwili t2, (t1<t2), to pierwszy zgłaszający się konsument otrzyma porcję wstawioną przez P(5), a następny przez P(2)). Do różnych elementów bufora procesy mogą mieć równoległy dostęp. Należy zadbać o to, aby później przybyły konsument mógł zacząć wcześniej pobierać z bufora, jeśli przeznaczona dla niego porcja będzie wcześniej gotowa.

4.3.5 Baza danych

Niektóre operacje pisania mogą prowadzić do zmiany struktury bazy danych. Takimi operacjami są wstawianie oraz usuwanie rekordu, natomiast modyfikacja istniejącego rekordu nie zmienia struktury bazy. Zmiana struktury powoduje konieczność przebudowy bazy danych. W tym celu proces wstawiający lub usuwający rekord musi poznać strukturę bazy, wykonując operację nazywaną intencją pisania. Może ona odbywać się równocześnie z operacjami odczytu lub operacją modyfikacji, ale wyklucza inne operacje intencji oraz wstawiania/usuwania.

Następująca tabelka przedstawia możliwość wykonania operacji na bazie danych. Litera c oznacza operację czytania, m - modyfikacji istniejącego rekordu, i - intencji, w -

wstawienia/usunięcia rekordu. Znak + oznacza możliwość jednoczesnego wykonywania operacji, a znak - wzajemne wykluczanie się operacji.



Napisz monitor synchronizujący dostęp do bazy danych procesów wykonujących albo operacje odczytu, albo modyfikacji, albo intencji poprzedzające wstawianie/usuwanie.

4.3.6 Prom jednokierunkowy

Napisz monitor synchronizujący pracę promu rzecznego i samochodów, które z niego korzystają. Prom wpuszcza N=10 samochodów lub, jeśli samochodów jest za mało, czeka aż upłynie T=30 min (procedura MON.WPUŚĆ). Następnie przepływa, na drugą stronę rzeki, wypuszcza samochody (procedura MON. WYPUŚĆ) i wraca (stąd określenie ednokierunkowy}. Samochody czekają na wjazd (procedura MON.CZEKAJ), wjeżdżają na prom, informują o wjechaniu i czekają na, zjazd (procedura MON .WJECHAŁEM), zjeżdżają z promu, informują o zjechaniu (procedura MON.ZJECHAŁEM). Wjeżdżanie i zjeżdżanie odbywa się pojedynczo.
Zakładamy istnienie procesu ZEGAR, który co minutę wywołuje procedurę monitora MON.CYK.

4.3.7 Trzy drukarki

Grupa procesów P(I..N) (N> 4) korzysta z trzech drukarek wierszowych LP-1,LP-2 i LP-3. Proces musi zarezerwować drukarkę przed rozpoczęciem korzystania z niej. Z i-1 ą drukarką jest związana zmienna drukuj e [i], której wartością jest numer procesu korzystającego z drukarki lub 0. Napisz monitor udostępniający procedury rezerwacji i zwalniania drukarki:
procedurę ZAJMIJ(i:nrprocesu; var dr:nrdrukarki);
i — numer procesu żądającego przydziału,
dr — numer drukarki przydzielonej procesowi,
procedurę ZWOLNIJ(dr:nrdrukarki);
dr — numer drukarki zwalnianej przez proces.

4.3.8 Lotniskowiec

Rozwiąż zadanie 3.3.11 używając mechanizmu monitora. Monitor powinien udostępniać procedury CHCĘ_STARTOWAĆ, WYSTARTOWAŁ, CHCĘ_LĄDOWAĆ i WYLĄDOWAŁ,

które będą wywoływane odpowiednio przed startem, po starcie, przed lądowaniem i po wylądowaniu samolotu.

4.3.9 Stolik dwuosobowy

Napisz monitor KELNER sterujący dostępem do stolika dwuosobowego. Ze stolika korzysta N mężczyzn i N odpowiadających im kobiet. Algorytm zarówno mężczyzny, jak i kobiety o numerze j jest następujący:

```
repeat
      własne_sprawy;
      KELNER.CHCĘ.STOLIK(j);
      randka;
      KELNER.ZWALNIAM
forever;
```

Stolik jest przydzielany jednocześnie kobiecie j i mężczyźnie j dopiero wtedy, gdy oboje go zażądają. Zwalnianie nie musi być jednoczesne (tzn. siadają przy stole razem, ale odejść mogą oddzielnie).

4.3.10 Zasoby dwóch typów

W systemie znajdują się dwa typy zasobów: A i B. które są wymienne, ale pierwszy z nich jest wygodniejszy niż drugi (np. szybszy). Jest M zasobów typu A oraz N (N>M) zasobów typu B. W systemie działają trzy grupy procesów, które różnią się sposobem zgłaszania zapotrzebowania na zasób.

Procesy pierwszej grupy żądają wyłącznie wygodnego zasobu i czekają, aż będzie dostępny. Procesy drugiej grupy żądają wygodnego zasobu, lecz jeśli jest niedostępny, to czekają na zasól) dowolnego typu. Procesy trzeciej grupy żądają zasobu dowolnego typu, lecz nie czekają, jeśli zasoby nie są dostępne.

Napisz monitor udostępniający procedury przydziału i zwalniania zasobów.

4.3.11 Algorytm bankiera

Napisz monitor BANKIER realizujący algorytm bankiera zarządzania M jednorodnymi jednostkami pewnego zasobu. Z zasobu korzysta N procesów, które żądają zawsze pojedynczych jednostek zasobu wywołując procedurę BANKIER.BIORĘ(i), przy czym i jest numerem procesu, a zwalniają jednostki pojedynczo wywołując procedurę BANKIER. ZWALNIAM (i). Wiadomo, że maksymalne potrzeby procesu i nie przekraczają f (i) jednostek, przy czym f (i) < M, i=1, ... ,N.

Algorytm bankiera polega na tym, że przed przydzieleniem jednostki zasobu sprawdza się, czy stan, w jakim znajdzie się system po przydzieleniu, będzie stanem bezpiecznym. Jeśli stan ten będzie bezpieczny, to zasób zostanie przydzielony, w przeciwnym razie proces żądający zasobu musi czekać.

Stan systemu jest bezpieczny, jeśli istnieje taka sekwencja zwrotów i przydziałów zasobów, że zaspokoi żądania wszystkich procesów.

W celu stwierdzenia, czy stan jest bezpieczny, system musi przechowywać następujące informacje. W tablicach potrzeby i przydział typu `array[l..N] of integer` są pamiętane

jeszcze nie zaspokojone potrzeby procesów oraz liczba już przydzielonych jednostek zasobu, natomiast w zmiennej dostępne typu integer pamięta się liczbę jeszcze dostępnych jednostek zasobu.

Funkcja dająca odpowiedź, czy stan po przydzieleniu jednej jednostki zasobu procesowi numer i jest bezpieczny wygląda następująco:

```
function bezpieczny(i: integer): boolean;
var całkowite, j: integer;
dalej: boolean;
koniec: array[l..N] of boolean; {procesy zakończone}
begin
if dostępne = O then bezpieczny := false
else begin
      przydział [i] := przydział[i] + 1;
                    {markowanie przydziału}
      potrzeby [i] := potrzeby ":l - 1; {potrzeby maleją}
      całkowite := dostępne -  ;    {tyle jest jednostek}
      for j := l to N do koniec [j] := przydział[j] = 0;
      repeat
      j := i;
      dalej := true;
      while dalej do
            if not koniec[j] and (potrzeby[j] <= całkowite)
            then begin              {potrzeby tego procesu
                                     można zaspokoić}
            dalej := false;      {nie trzeba już szukać}
            całkowite := całkowite + przydział[j];
                                  {proces ten zwróci
                                   wszystko, co miał}
      koniec[j]:= true     {i będzie zakończony}
            end
            else begin          {tego procesu nie można
            j := j + 1;         teraz zaspokoić}
            if j > N then dalej := false
            end
      until (całkowite = M) or (j > N):
      bezpieczny := całkowite = M;   {odzyskano wszystkie}
      przydział[i] := przydział [i] - l;{powrót do}
      potrzeby[i] := potrzeby[i] + l  {starych wartości}
      end
end;
```

Powyżej opisano algorytm bankiera w przypadku pojedynczych żądań i jednego typu zasobu. Idea algorytmu dla wielu typów zasobów i jednoczesnych żądań wielu jednostek różnych zasobów jest taka sama z tym, że bardziej złożone są struktury danych (por. [PeSGOl]).

4.3.12 Rodzina procesów

Zarządzanie rodziną procesów polega na tworzeniu i niszczeniu procesów potomnych. Początkowo w systemie znajduje się jeden proces-antenat. Każdy proces może utworzyć wiele procesów potomnych. Potomny może na swoje żądanie zostać zniszczony, jeśli sam nie ma procesów potomnych. W przeciwnym razie musi zaczekać na samozniszczenie wszystkich swoich procesów potomnych.

Każdy proces ma numer, będący liczbą naturalną nie większą od N (w systemie może być co najwyżej N procesów). Tworzy się je wywołując systemową procedurę utwórz(n), a niszczy procedurą systemową zniszcz(n). Parametr n jest numerem procesu, który ma być utworzony (zniszczony).

Napisz monitor udostępniający funkcję NARODZINY i procedurę ŚMIERĆ.

Funkcja NARODZINY (n), wywoływana w procesie o numerze n ma powodować utworzenie dla niego procesu potomnego, jeśli nie będzie przekroczony limit procesów. Funkcja ta powinna zwracać numer utworzonego procesu lub O, gdy procesu nie można utworzyć.

Procedura ŚMIERĆ (n) ma powodować samozniszczenie procesu o numerze n (jeśli jest to numer procesu-antenata, to operacja ta ma efekt pusty).

4.3.13 Szeregowanie żądań do dysku

Dysk magnetyczny jest dzielony przez wiele procesów generujących żądania wejścia-wyjścia. Żądania są scharakteryzowane następującymi parametrami (C oznacza liczbę cylindrów dysku):

```
type par.żądania = record
      cylinder: l..C;
      sektor:  integer;
end;
```

Wykonanie żądania obejmuje przesunięcie głowic nad żądany cylinder, opóźnienie rotacyjne aż żądany sektor znajdzie się pod głowicą oraz transmisję danych między dyskiem a pamięcią operacyjną. Żądania są wykonywane synchronicznie. Proces żąda dostępu do dysku za pomocą procedury PRZYDZIEL, wykonuje żądanie, a potem zwalnia dysk wywołując procedurę ZWOLNIJ.

W celu zmniejszenia średniego czasu przesunięcia głowic stosuje się różne strategie szeregowania żądań wejścia-wyjścia. Jedną z nich jest SCAN następne do obsługi jest wybierane żądanie, spełniające następujące dwa warunki:

a) dotyczy cylindra najbliższego w stosunku do cylindra bieżącego,

b) przesunięcie głowic konieczne do obsługi tego żądania będzie odbywało się w tym samym kierunku, co poprzednie przesunięcie. Jeżeli nie ma, żądania, spełniającego drugi warunek, to jest wybierane żądanie spełniające tylko pierwszy warunek.

Napisz monitor udostępniający procedury PRZYDZIEL i ZWOLNIJ.
Wersja 1. Żądania są szeregowane zgodnie ze strategią FCFS (First Coine First Scrved}.
Wersja 2. Żądania są szeregowane zgodnie ze strategią SCAN.

4.3.14 Asynchroniczne wejście-wyjście

Procesy generują żądania, wejścia-wyjścia, które mają, być wykonywane asynchronicznie i w kolejności wygenerowania. Proces żądający wejścia-wyjścia wywołuje proceduro. ŻĄDANIE_WE_WY (żądanie: par .żądania). Po jej wykonaniu kontynuuje obliczenia, aż będą mu potrzebne wyniki wykonania żądania, wtedy wywołuje procedurę CZEKAM_NA_WYKONANIE(żądanie :par_żądania).

Inicjację wykonania, przez dysk wybranego żądania powoduje procedura start_dysku (żądanie : par .żądania).

Po zrealizowaniu żądania dysk generuje przerwanie wejścia-wyjścia. które powoduje wykonanie procedury PRZERWANIE_WE_WY.

Napisz monitor zarządzający wejściem-wyjściem, który będzie udostępniał procedury: ŻĄDANIE_WE_WY, PRZERWANIE_WE_WY, CZEKAM_NA_WYKONANIE.

4.3.15 Dyskowa pamięć podręczna

W systemie znajduje się intensywnie używany dysk i przeznaczona dla niego pamięć podręczną (cache) wielkości R ramek. W jednej ramce może być przechowywany jeden sektor dyskowy. Na dysku jest zapisana stała baza danych, z której dane są tylko odczytywane.

Pamięć podręczna jest zarządzana zgodnie ze strategią LRU (Lcast Recently Used], to znaczy w przypadku braku miejsca jest usuwany z niej sektor najdawniej używany.

Żądania odczytu są wykonywane synchronicznie, tzn. proces jest wstrzymywany, aż żądany sektor znajdzie się w pamięci operacyjnej. Jeżeli znajduje się on w pamięci podręcznej, to jest transmitowany stamtąd do pamięci operacyjnej. Żądania odczytu są realizowane przez procedurę

```
ŻĄDANIE_ODCZYTU(żądanie: par_żądania; adresPO : integer).
```

Jeżeli podczas jej wykonania okaże się, że żądanego sektora nie ma w pamięci podręcznej, to zostaje wywołana procedura

```
ŻADANIE_WE_WY (żądanie :par_żądania),
```

udostępniana przez monitor DYSK (patrz zadanie 4.3.14), szeregująca żądania do dysku i inicjująca wczytanie sektora o podanym adresie z dysku do pamięci podręcznej. Wczytanie sektora odbywa się asynchronicznie, tzn. powrót z tej procedury jest natychmiastowy. Proces żądający wczytania sektora będzie wstrzymany w procedurze ŻĄDANIE.ODCZYTU. Po wczytaniu sektora do pamięci podręcznej, podczas obsługi przerwania wejścia-wyjścia, jest wywoływana procedura KONIEC_ODCZYTU, która wznawia proces oczekujący na wczytanie tego sektora.

Transmisję sektora z ramki ramka z pamięci podręcznej do pamięci operacyjnej, poczynając od adresu, przekazywanego jako parametr adresPO do procedury ŻĄDANIE_ODCZYTU, wykonuje synchronicznie procedura systemowa zPPdoPO(sektorPP,adresPO:integer).

Napisz monitor udostępniający procedurę ŻĄDANIE_ODCZYTU i procedurę KONIEC_ODCZYTU.

4.3.16 Pamięć operacyjna — strefy dynamiczne

Pamięć operacyjna o pojemności rozmiar słów, adresie początkowym O, jest dzielona dynamicznie na strefy przydzielane zgodnie ze strategią First Fit (tzn. jest przydzielana pierwsza w kolejności strefa o wystarczającej wielkości). Jeżeli żądanie nie może być zrealizowane z powodu braku strefy odpowiedniej wielkości, to jest wstrzymywane. Jeśli jakieś żądanie jest wstrzymane, to żądania późniejsze są również wstrzymywane. Żądania

wstrzymane są wznawiane zgodnie ze strategią FCFS (tzn. są wznawiane w takiej kolejności, w jakiej zostały wstrzymane).

Napisz monitor zarządzający pamięcią operacyjną, udostępniający procedury:

```
PRZYDZIEL(var adres: integer; ile: integer),
```

która przydziela zadaniu strefę wielkości ile i zwraca adres adres początku przydzielonej strefy, oraz

`ZWOLNIJ (adres : integer)`, zwalniającą strefę o adresie adres.

4.3.17 Strategia buddy

(Uwaga: nie mylić ze strategią Buddy — założyciela buddyzmu. Nazwa omawianej tu strategii pochodzi od angielskiego słowa buddy, co znaczy kumpel.)

W systemie wieloprocesorowym pamięć operacyjna jest zarządzana zgodnie ze strategią buddy. Jednostką przydziału pamięci jest obszar wielkości c*2, gdzie c jest wielkością bloku pamięci, a i jest dobierane w należności od żądanej wielkości pamięci. Wielkość całej pamięci wynosi c*2A .. Początkowo cała pamięć jest wolna. W zależności od potrzeb pamięć tę dzieli się na polowy, a te znów na polowy itd., umieszczając je na listach wolnych obszarów oddzielnych dla każdej wielkości.

Z chwilą pojawienia się żądania przydziału pamięci o rozmiarze r > c (przy czym c*2*c^1 < r < c* 2^2, k > 0) szuka się wolnego obszaru wielkości c*2^c; natomiast dla żądania przydziału obszaru o rozmiarze r < c, szuka się wolnego obszaru wielkości c. Jeśli takiego obszaru nie ma, bierze się najmniejszy większy obszar o wielkości c. * 2% i > k i dzieli się go odpowiednią liczbę razy na polowy, aż do uzyskania obszaru potrzebnej wielkości. Pozostałe połówki umieszcza się na listach wolnych obszarów.

Z chwilą zwolnienia obszaru o wielkości r:*2^ na liście wolnych obszarów o tej wielkości szuka się obszaru, który jest jego kumplem, tzn. powstał razem z nim w wyniku podziału większego obszaru. Jeśli taki obszar znajdzie się, oba łączą się ponownie w jeden obszar dwa razy większy. Następnie szuka, się kumpla dla tego większego obszaru, itd. Jeśli dla zwolnionego obszaru nie ma jego drugiej wolnej połówki, obszar ten jest umieszczany na liście wolnych obszarów.

Założymy, że wskaźniki do list wolnych obszarów o rozmiarach c * 2k są pamiętane w tablicy głowy [k] , k = 0. .K. Na listach tych można wykonywać operacje, którym odpowiadają następujące procedury: procedurę `usuń(k:O..K; var a:adres)` — usuwa pierwszy obszar z listy k zwracając jego adres a, procedurę `wstaw(k:0. .K; a:adres)` — dodaje do listy k obszar o adresie a, `function znajdź_usuń(k:0..K; a:adres) :boolean` — odpowiada na pytanie, czy na liście k jest obszar o adresie a, jeśli tak, to go z niej usuwa.

W systemie tym działają procesy, które intensywnie alokują i zwalniają obszary pamięci różnej wielkości. Aby maksymalnie zrównoleglić działanie systemu, każdą listą wolnych obszarów o numerze k zarządza oddzielny monitor BUDDY (k). Monitor taki udostępnia procesom procedury PRZYDZIEL i ZWOLNIJ. Jeśli brakuje wolnego obszaru wielkości c*2fc, w wyniku wywołania procedury systemowej `start_dziel(k+1)` uruchamia się `proces DZIEL(k-H)`, którego zadaniem jest pozyskanie wolnego obszaru o wielkości c * 2fc+1, podzielenie go na połowy o adresach al, a2, a następnie wywołanie w monitorze BUDDY (k) procedury `KONIEC_DZIEL (al ,a2: adres).`

Jeśli odzyskany obszar o adresie al można połączyć z jego kumplem o adresie a2, to w wyniku wywołania `procedury start_Łącz(k+1 ,al ,a2)` powołuje się do życia proces `ŁĄCZ (k+1 ,al,a2)`, którego jedynym zadaniem jest zwrócenie odpowiedniego obszaru.

Napisz monitor BUDDY (k :0..K) zarządzający k-tą listą oraz procesy DZIEL (k:l..K) i LĄCZ(k:l..K; al,a2:adres).

4.4 Rozwiązania

4.4.1 Trzy grupy procesów

W tym zadaniu szczególną uwagę trzeba zwrócić na procesy trzeciej grupy, które łatwo mogą zostać zagłodzone przez pozostałe procesy. Z tego względu procesy trzeciej grupy powinny być uprzywilejowane podczas przydzielania zasobów i zwalniania ich przez procesy dwóch pierwszych grup. Przy takim rozwiązaniu powstaje jednak pytanie, kiedy należy wznawiać procesy żądające pojedynczych zasobów. Jeżeli proces trzeciej grupy oddając zasoby będzie wznawiał również proces tej grupy, to procesy żądające pojedynczych zasobów mogą zostać zagłodzone. Dlatego proces trzeciej grupy oddając zasoby usiłuje najpierw wznowić procesy dwóch pierwszych grup.

Zmienne wolneA i wolneB służą do pamiętania liczby wolnych zasobów odpowiedniego typu, zmienne mamA i mamB do rezerwacji zasobów dla pierwszego czekającego procesu grupy trzeciej.

```
  type typ_zasobu = (A, B, AB);

  monitor AiB;
  const M = ?;           {liczba zasobów typu A>
        N = ?;           {liczba zasobów typu B}

  var KOLEJKA.A, KOLEJKA.B, KOLEJKA.AB: condition;   {do wstrzymywania procesów}
    wolneA, wolneB: integer;   {liczby wolnych zasobów}
    mamA, mamB: boolean;     {do rezerwacji zasobów}

  export procedure ŻĄDANIE(typ: typ_zasobu);
  begin
   case typ of
   A: begin
     if (wolneA = 0) or    {brak A lub jest potrzebny}
       (not empty(KOLEJKA_AB) and not mamA) then
       wait(KOLEJKA.A);
     wolneA := wolneA - l
    end;
   B: begin
     if (wolneB = 0) or    {brak B lub jest potrzebny}
       (not empty(KOLEJKA_AB) and not mamB) then
       wait(KOLEJKA.B);
     wolneB := wolneB - l
    end;
AB: begin
   if wolneA * wolneB = O then begin
    if empty(kolejka.AB) then begin
     mamA := wolneA > 0; {rezerwacja zasobów}
     mamB := wolneB > O  {dla siebie}
```

```
    end;
    wait(KOLEJKA.AB);
   end;
   wolneA := wolneA - 1;
   wolneB := wolneB - 1;
   if not empty(kolejka_AB) then begin
    mamA := wolneA > 0;  {rezerwacja zasobów}
    mamB := wolneB > O   {dla następnego w kolejce}
   end
  end
 end
end; {ŻĄDANIE}

export procedure ZWOLNIENIE(typ: typ.zasobu) ;
begin
 case typ of
 A: begin
   wolneA := wolneA + 1;
   if not empty(KOLEJKA_AB) and not mamA then
    if mamB then      {miał B, brakowało A}
     signal(KOLEJKA_AB)  {może brać oba}
    else          {nie miał A ani B}
     mamA := true     {już ma A}
   else           {A nie potrzebny 3. gr.}
    signal(KOLEJKA_A)   {mogą go brać z l.gr.}
  end;
 B: begin
   wolneB := wolneB + 1;
   if not empty(KOLEJKA_AB) and not mamB then
    if mamA then      {miał A, brakowało B}
     signal(KOLEJKA_AB)  {może brać oba}
    else          {nie miał A ani B}
     mamB := true     {już ma B}
   else           {B nie potrzebny 3. gr.}
    signal(KOLEJKA_B)   {mogą go brać z 2. gr.}
  end;
  AB: begin
     wolneA := wolneA + 1;
     wolneB := wolneB + 1;
     if empty(KOLEJKA_A) and empty(KOLEJKA_B) then
      signal(KOLEJKA_AB)
     else begin         {pojedyncze mają priorytet}
      signal(KOLEJKA_A);
      signal(KOLEJKA_B)
     end
    end
   end
  end; {ZWOLNIENIE}
  begin
   wolneA := M;
   wolneB := N;
   mamA := false;
   mamB := false
  end; {AiB}
```

Warto zwrócić uwagę na fakt, że zadanie to jest innym sformułowaniem problemu jednoczesnej operacji opuszczania dwóch semaforów Przedstawione tu rozwiązanie jest znacznie prostsze niż rozwiązanie tego problemu za pomocą semaforów (por. 3.4.4). Nie trzeba dbać o wzajemne wykluczanie przy dostępie do wspólnych zmiennych zapewnia to sam

monitor), nie trzeba też liczyć wstrzymanych procesów, gdyż interesuje nas tylko, czy w ogóle jakieś procesy czekają, a na to pytanie daje nam odpowiedź funkcja empty.

4.4.2 Przetwarzanie potokowe

Tablica ile służy do pamiętania liczby porcji, które może przetworzyć proces, tablica warunków C zaś do oddzielnego wstrzymywania każdego procesu.

```
monitor DAJWEŹ;
const N = ?;
var ile: array [O..N-1] of integer;
      C: array[0..N-1] of condition;

export procedure WEŹ(i: integer);
begin
 if ile [i] = O then wait(C[i]);
 ile[i] := ile [i] - l
end; {WEŹ}
export procedure DAJ(i: integer);
begin
 ile[(i + 1) mod N] := ile[(i + 1) mod N] + 1;
 signal(C[(i + 1) mod N])
end; {DAJ}
begin
 ile[0] := M;
 for i:=l to N - l do ile[i]:=0
end; {DAJWEŹ}
```

Zauważmy, że monitor DAJWEŹ jest jednoczesną implementacją N semaforów. Zmienna ile[i] służy do pamiętania wartości semafora, warunek C[i] do wstrzymywania procesów. Procedura WEŹ odpowiada operacji P na semaforze, a procedura DAJ operacji V na semaforze. Rozwiązanie to odpowiada więc dokładnie rozwiązaniu zadania 3.3.7 o linii produkcyjnej.

4.4.3 Producenci i konsumenci z losową wielkością wstawianych i pobieranych porcji

Jeśli proces żąda wolnych lub zajętych elementów bufora w liczbie przekraczającej aktualny stan, to musi być wstrzymany. Wznowienie może nastąpić dopiero wtedy, gdy w buforze znajdzie się odpowiednia liczba wolnych lub zajętych elementów. Zauważmy jednak, że jeśli zdecydujemy się wznawiać proces natychmiast, gdy tylko bufor osiągnie oczekiwany stan, to możemy doprowadzić do zagłodzenia procesów mających duże wymagania. Może się np. tak zdarzyć, że nigdy nie będzie N wolnych elementów w buforze, mimo iż konsumenci będą pobierać porcje z bufora, gdyż w tym samym czasie bufor będą zapełniać „drobni" producenci.

W prezentowanym rozwiązaniu zakłada się, że procesy są obsługiwane w takiej kolejności, w jakiej zgłosiły swe żądania. Pierwszy proces w kolejce jest wstrzymywany oddzielnie na zmiennej PIERWSZYPROD lub PIERWSZYKONS. Czeka on, aż bufor osiągnie wymagany stan, przy czym odpowiedni warunek sprawdza się po każdej zmianie stanu bufora. Takie podejście chroni przed zagłodzeniem, zmniejsza jednak wykorzystanie bufora, gdyż

procesy są wstrzymywane czasem także wtedy, gdy w buforze jest wymagana liczba wolnych lub zajętych elementów.

```
monitor BUFOR;
const M = ?;
var buf: array[l..2*M] of element;  {bufor 2M-elementowy»
   jest: integer;   {liczba porcji w buforze}
  PIERWSZYPROD, RESZTAPROD, PIERWSZYKONS, RESZTAKONS: condition;
export procedure WSTAW(ile: integer; p: porcja);
begin
 if not empty(PIERWSZYPROD) then wait(RESZTAPROD);
            {inny producent też już czeka}
 while 2*M - jest < ile do wait(PIERWSZYPROD);
            {czekanie na "ile" wolnych miejsc}
 jest := jest + ile;
 do_bufora(ile,p);
 signal(RESZTAPROD);  {pierwszy z reszty będzie pierwszym}
 signal(PIERWSZYKONS) {pierwszy konsument sprawdzi stan}
end; {WSTAW}

export procedure POBIERZ(ile: integer; var p: porcja);
begin
 if not empty(PIERWSZYKONS) then wait(RESZTAKONS);
            {inny konsument już czeka}
 while jest < ile do wait(PIERWSZYKONS);
            {czekanie na żądaną liczbę porcji}
 jest := jest - ile;
 z_bufora(ile,p);
 signal(RESZTAKONS);  {pierwszy z reszty będzie pierwszym}
 signal(PIERWSZYPROD) {pierwszy producent sprawdzi stan}
end; {POBIERZ}
begin
 jest := O
end; {BUFOR}
```

Ponieważ nie może być takiej sytuacji, że jednocześnie czekają producenci i konsumenci (byłaby wówczas blokada, a wyklucza ją założenie, że liczba żądanych elementów bufora nie przekracza jego połowy), zatem w procesie wznowionym po oczekiwaniu na warunek PIERWSZYPROD lub PIERWSZYKONS drugie signal zawsze „pójdzie w powietrze". Podobnie w procesie, który bez czekania wykonuje procedurę monitora, zawsze pierwsze signal „pójdzie w powietrze".

4.4.4 Producenci i konsumenci z asynchronicznym wstawianiem i pobieraniem

Rozwiązanie wersji I

Monitor BUFOR chroniący dostępu do bufora, którego elementy są numerowane od O do N - 1 musi udostępniać cztery procedury: rozpoczynające operacje wstawiania i pobierania oraz kończące je. Dwie pierwsze będą zwracały numer elementu bufora, na którym proces może zacząć działać, parametrem dwóch następnych będzie numer elementu bufora, na którym proces właśnie skończył działać. Każdy element bufora może znajdować się w jednym z trzech stanów: pustym, zajętym i pełnym. Stan pusty oznacza, że element można przydzielić producentowi, pelny, że można go przydzielić konsumentowi, zajęty, że element jest w trakcie zapełniania. (Ponieważ jest tylko jeden konsument, nie potrzeba wyróżniać stanu opróżniania.) Zmienna ZABLOKOWANY służy do wstrzymywania konsumentów (producenci

nie są tu wstrzymywani). Wskaźniki do_włożenia i do .wyjęcia wskazują odpowiednio miejsce, do którego trzeba wstawić porcję, i miejsce, z którego trzeba pobrać porcję.

```
monitor BUFOR;
const N = ?;
var do_wstawienia, do_pobrania: O..N-1;
  stan: array[0..N-l] of (pusty, zajęty, pełny);
  ZABLOKOWANY: condition; {do wstrzymywania konsumentów}
  i: integer;       {pomocnicza do inicjacji}

 export procedure WSTAW_POCZ(var i: O..N);
begin
  if stan[do_wstawienia] <> pusty then
              {nie ma wolnych miejsc}
  i := N          {porcja będzie stracona}
  else begin         {wolne miejsce będzie zajęte}
  i := do.wstawienia;
  stan[do_wstawienia] := zajęty;
  do.wstawienia := (do.wstawienia + 1) mód N
  end
 end; {WSTAW.POCZ}

 export procedure WSTAW_KONIEC(i: O..N-1);
 begin
  stan[i] := pełny;
  if i = do.pobrania then signal(ZABLOKOWANY)
 end; {WSTAW.KONIEC}

 export procedure POBIERZ_POCZ(var i: O..N-1);
 begin
  if stan[do_pobrania] O pełny then wait(ZABLOKOWANY);
  i := do.pobrania;
  do.pobrania := (do.pobrania + 1) mód N
 end; {POBIERZ.POCZ}
export procedure POBIERZ_KONIEC;
begin                              j
 stan[do_pobrania] := pusty
end; {POBIERZ.KONIEC}

begin                              j
 do_wstawienia := 0;
 do_pobrania := 0;
 for i := O to N - l do stan[i] := pusty
end; {BUFOR}
```

Rozwiązanie wersji II

W rozwiązaniu zadania w tej wersji trzeba kolejkować pełne i puste elementy bufora, aby przydzielać je we właściwej kolejności. Przyjmiemy zatem, że jest dostępny specjalny typ kolejka oznaczający kolejkę liczb całkowitych i że na zmiennych tego typu można wykonywać operacje wstawiania, do kolejki (procedura do_kolejki), usuwania z niej (procedura z_kolejki) i sprawdzania, czy jest pusta (funkcja pusta_kol). Do wstrzymywania producentów i konsumentów oczekujących na przydział elementu bufora służą zmienne CZEK_PROD i CZEK_KONS. Tablica PRZYDZ_KONS służy do wstrzymywania konsumentów po przydzieleniu elementu bufora, ponieważ muszą oni jeszcze poczekać na jego wypełnienie.

```
monitor BUFOR;
const N = ?
```

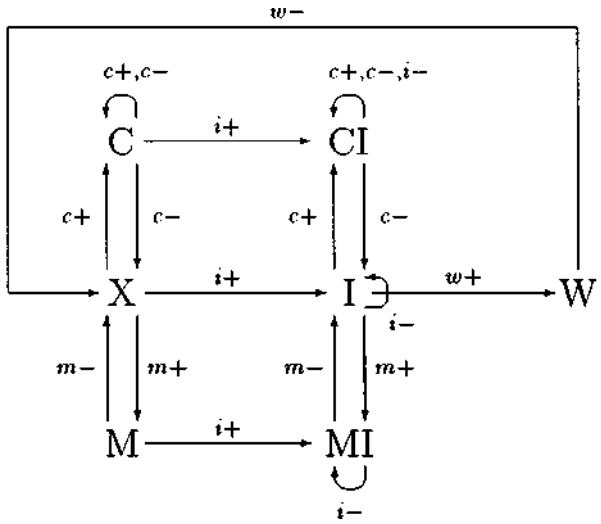
```
 var CZEK.PROD,        {do wstrzymywania producentów}
   CZEK_KONS,        {i konsumentów bez przydziału}
   PRZYDZ_KONS: condition; {i konsumentów z przydziałem}
   puste, pełne: kolejka; {kolejki pustych i pełnych elementów bufora}
   jest.porcja: array[0..N-l] of boolean; {porcja w elemencie bufora jest
                gotowa do konsumpcji}
   j: integer;       {do inicjacji kolejki puste}

 procedure do_kolejki(k: kolejka; i: integer);
  {wstawia liczbę i na koniec kolejki k}
 procedure z_kolejki(k: kolejka; var i: integer):
  {usuwa z kolejki k pierwszy element i podstawia go na i}
 function pusta_kol(k: kolejka): boolean;
  {true, gdy kolejka k jest pusta}
export procedure WSTAW.POCZ(var i: O..N-1);
begin
 if pusta_kol(puste) then wait(CZEK_PROD);
 z_kolejki(puste, i);    {element i-ty będzie teraz}
 do.kolejki(pełne, i);   {zapełniany i, mimo że}
 jest_porcja[i] := false; {nie ma w nim jeszcze porcji,}
 signal(CZEK_KONS)     {można go dać konsumentowi}
end;

export procedure WSTAW_KONIEC(i: O..N-1);
begin
 jest_porcja[i] := true;  {już jest porcja w elemencie i}
 signal(PRZYDZ_KONS[i])   {konsument może ją wziąć}
end;

export procedure POBIERZ_POCZ(var i: O..N-1);
begin
 if pusta.kol(pełne) then wait(CZEK_PROD);
 z_kolejki(pełne, i);    {element i-ty opróżni się}
 if not jest_porcja[i] then wait(PRZYDZ_KONS[i]);
end;

export procedure POBIERZ_KONIEC(i: O..N-1);
begin
 do.kolejki(puste, i);   {element i-ty już opróżniony}
 signal(CZEK_PROD)     {może go wziąć producent}
end;

begin
 for i := O to N - l do do_kolejki(puste, i)
end;
```

4.4.5 Baza danych

Operacja intencji i następująca po niej operacja wstawiania/usuwania wykonywane przez jeden proces nie mogą być rozdzielone operacjami intencji lub wstawiania/usuwania pochodzącymi od innych procesów. Operacja wstawiania/usuwania zmienia bowiem strukturę bazy danych. Ponieważ operacja wstawiania/modyfikacji nie musi następować bezpośrednio po operacji intencji, więc dopuścimy wykonywanie operacji czytania lub modyfikacji między nimi. Ponadto założymy, że w procesach operacje intencji i wstawiania/usuwania zawsze tworzą parę i występują w tej kolejności.

Baza danych może być w jednym z następujących stanów:
X — żadna operacja nie jest wykonywana,
C — są wykonywane operacje czytania,
W — jest wykonywana operacja wstawiania/usuwania,
M — jest wykonywana operacja modyfikacji,
I — jest lub była wykonywana operacja intencji,
CI — są wykonywane operacje czytania i jest lub była wykonywana operacja intencji,
MI — jest wykonywana operacja modyfikacji i jest lub była wykonywana operacja intencji.

Przejścia pomiędzy stanami bazy danych przedstawia rys. 4.2. Symbole c+, i+, w+, m+ oznaczają odpowiednio rozpoczęcie operacji czytania, intencji, modyfikacji, wstawiania/usuwania, natomiast c—, i—, m—, w— ich zakończenie. Przejścia te pokazują, że jest niemożliwe wykonanie kolejnych operacji intencji bez wykonania operacji wstawiania/usuwania między nimi.
Jeżeli baza danych jest w stanie, w którym nie można rozpocząć wykonywania żądanej operacji, to proces jest wstrzymywany.



RYS. 4.2. Zmiany stanu bazy danych

Proces żądający wstawiania/usuwania ma najwyższy priorytet. Czeka on jedynie na zakończenie trwających już operacji czytania lub modyfikacji.

Aby uniknąć zagłodzenia, pewne procesy są wstrzymywane, gdy czekają już procesy innej grupy. Proces żądający czytania jest wstrzymywany, gdy czekają procesy żądające modyfikacji lub wstawiania/usuwania. Proces żądający modyfikacji jest wstrzymywany, gdy czekają procesy żądające czytania lub wstawiania/usuwania. Procesy te nie muszą sprawdzać, czy czekają procesy, które chcą wykonywać operację intencji, gdyż nie wykluczają się z nimi.

Wznawianie procesów odbywa się tak, aby uniknąć zagłodzenia. Po czytaniu jest wznawiany proces czekający na wstawianie/usuwanie, a jeśli go nie ma, to proces czekający na modyfikację. Po zakończeniu modyfikacji jest wznawiany proces czekający na wstawianie/usuwanie, jeśli go nie ma, to procesy czekające na czytanie, a jeśli i tych nie ma, to kolejny proces czekający na modyfikację. Natomiast po zakończeniu wstawiania/usuwania żaden rodzaj procesów nie jest faworyzowany. W sytuacjach konfliktowych, gdy są jednocześnie procesy czekające na czytanie i na modyfikację, decyzja o rodzaju wznawianych procesów zależy od tego, jakie procesy były zwolnione po zakończeniu poprzedniego

wstawiania/usuwania. Do przechowywania informacji o tym służy zmienna wznowiono_C, która przyjmuje wartość true, gdy poprzednio były wznowione procesy czekające na czytanie. Poza tym próbuje się wznowić proces czekający na wykonanie operacji intencji.

Zakończenie operacji intencji nie powoduje wznowienia żadnego procesu, gdyż żądanie wstawiania/usuwania nie było jeszcze zgłoszone (przypominamy, że może ono pochodzić tylko od procesu, który kończy właśnie wykonywanie operacji intencji). Operacje czytania lub modyfikacji natomiast mogły być wykonywane jednocześnie z operacją intencji.

```
  type typ_żądania = (c, i, w, m);

  monitor BAZA.DANYCH;
  type stan.bazy = (X, C, I, W, M, CI, MI);
  var stan: stan_bazy;
    czyta: integer;   {liczba czytających czytelników}
    wznowiono_C: boolean;
    DO.C, DO_I, DO_M, DO_W : condition;

  export procedure ŻĄDANIE(s: typ.żądania);
  begin
   case s of
   c: begin
      if (stan in [W, M, MI]) or
        not empty(DO.M) or not empty(DO_W)
      then wait(DO_C);
      czyta := czyta + 1;
      if stan = X then stan := C
      else
     if stan = I then stan := CI;
         {w pozostałych przypadkach stan ten sam}
   end;
 m: begin
    if (stan in [C, M, W, CI, MI]) or
      not empty(DO_C) or not empty(DO_W)
    then wait(DO_M);
    if stan = X then stan := M
          else stan := MI;
   end;
  i: begin
    if stan in [I, W, CI, MI] then wait(DO_I);
    if stan = X then stan := I
    else
     if stan = C then stan := CI
           else stan := MI;
   end;
  w: begin
    if stan in [CI, MI] then wait(DO.W);
    stan := W
   end
  end
end; {ŻĄDANIE}

export procedure ZWOLNIENIE(s: typ_żądania);
var signal_C, signal.M: boolean;
begin
 case s of
 c: begin
    czyta := czyta - 1;
    if czyta = 0 then
     if stan = CI then stan := I
           else stan := X;
```

```
     if not empty(DO_W) then signal(DO_W)
               else signal(DO_M)
   end;
 m: begin
    if stan = MI then stan := I
          else stan := X;
    if not empty(DO_W) then signal(DO_W)
    else if not empty(DO.C) then
     while not empty(DO_C) do signal(DO_C)
     else signal(DO_M)
    end;
  w: begin
     stan := X;
     signal_C := empty(DO_M) or not wznowiono_C;
     signal_M := empty(DO_C) or wznowiono_C;
     wznowiono.C := signal.C and not empty(DO_C);
     if signal_C then
      while not empty(DO_C) do signal(DO_C);
     if signal.M then signal(DO_M);
     signal(DO_I)
    end
  end
  end; {ZWOLNIENIE}

  begin
  stan := X;
  czyta := 0;
  wznowiono_C := true
  end; {BAZA.DANYCH}
```

4.4.6 Prom jednokierunkowy

W monitorze użyto dwóch zmiennych do liczenia samochodów. Zmienna, n wskazuje liczbę samochodów dopuszczonych do wjechania na prom, zmienna k zaś liczbę samochodów już znajdujących się na promie. Ponieważ samochody wjeżdżają pojedynczo więc k może być co najwyżej o jeden mniejsze niż n. Prom może odpłynąć tylko wtedy, gdy k=n. Ważne jest więc, aby po upływie czasu postoju, poczekać na wjeżdżający jeszcze samochód. Relacja n=N oznacza, że prom jest niedostępny, tzn. za chwile może być pełny albo właśnie płynie, wypuszcza samochody, albo wraca. Na czas przepływania musi być wyłączony zegar, co uzyskuje się instrukcją t := T.

```
  monitor PROM;
  const N = 10;
    T = 30;
  var t: integer; {czas bieżący}
    n: integer;     {samochody wpuszczone na prom}
    k: integer;     {liczba samochodów już na promie}
    STOPI: condition;  {wstrzymuje samochody wjeżdżające}
    STOP2: condition;  {wstrzymuje samochody zjeżdżające}
    CUMA: condition;   {wstrzymuje prom}
export procedure CYK;
begin
 t := t + 1;
 if t = T then
   signal(CUMA)    {odjazd}
end; {CYK}
```

```
export procedure CZEKAJ;
begin
 if (k <> n) or (n = N)  {ktoś wjeżdża przed nami}
 then wait(STOPI);    {lub prom pełny}
  n := n + l              {wjeżdżam}
end; {CZEKAJ}

export procedure WJECHAŁEM;
begin
  k := k + 1;
  if n = N then          {prom pełny}
    signal(CUMA)         {może odpływać}
  else
    signal(STOPI);        {ktoś może wjechać}
  wait(STOP2)            {czekam na zjazd}
end; {WJECHAŁEM}

export procedure ZJECHAŁEM;
begin
  k := k - 1;
  if k > O then
   signal(STOP2)        {następny zjeżdża}
  else signal(CUMA)      {prom może wracać}
end; {ZJECHAŁEM}

export procedure WPUŚĆ;
begin
  t := 0;                {odliczamy czas}
  n := 0;                {na promie pusto}
  signal(STOPI);         {niech wjeżdżają}
  while k =  O do         {czekam na pełny prom}
    wait(CUMA);          {lub upływ czasu}
  t := T;                {wyłączam zegar}
  if k <> n  then         {upłynął czas, ale ktoś wjeżdża}
  begin
       n  := N;              {blokuję resztę}
       wait(CUMA)           {czekam, aż  wjedzie}
      end
      else n  :=  N
    end; {WPUŚĆ}

    export procedure WYPUŚĆ;
    begin
      signal(STOP2);
      wait(CUMA)
    end; {WYPUŚĆ}

    begin
      t := T;                {zegar wyłączony}
      n := N;                {prom niedostępny}
      k := O                 {nikt nie  wjeżdża}
    end;   {PROM}
```

Zauważmy, że ponieważ prom jest jednokierunkowy, więc powinien czekać w procedurze WPUŚĆ na wjechanie co najmniej jednego samochodu.

4.4.7 Trzy drukarki

Zmienna ZAJĘTY służy do wstrzymywania procesów, gdy wszystkie drukarki są zajęte.

```
monitor OBSŁUGA.DRUKAREK;
const  N =  ?;

var drukuje: array[1..3] of  O..N;  {nr drukującego procesu}
    ZAJĘTY:  condition;

export procedure ZAJMIJ(i:  1..N;  var  dr: 1..3);
var j:  integer;
begin
  if drukuje[1] * drukuje [2]  *  drukuje[3]
  then wait(ZAJĘTY);              {wszystkie drukują}
   j:=l;                            {szukam wolnej}
   while drukuje[j] O O  do
     j  :=  j + 1;
   drukuje[j] := i;
   dr := j
end;
export procedure ZWOLNIJ(dr: 1..3);
begin
  drukuje[dr]  := O;
  signal(ZAJĘTY)
end;

begin
  drukuje[1]:=  0;                {nikt nie  drukuje}
  drukuje[2]:=  O;
  drukuje[3]:=  O
end;  {OBSLUGA_DRUKAREK}
```

4.4.8 Lotniskowiec

W tym rozwiązaniu rolę semaforów PAS_DLA_START i PAS_DLA_LĄD z rozwiązania 3.4.11 pełnią zmienne typu condition DO_STARTU i DO_LĄDOWANIA. Funkcję semafora DOSTĘP spełnia sam monitor LOTNISKOWIEC. Nie są też potrzebne liczniki chce_start i chce_ląd, wystarczy w tym przypadku sprawdzenie, czy odpowiednia kolejka jest pusta. Warto zauważyć, że procedura zwolni j _pas nie jest dostępna na zewnątrz monitora.

```
monitor LOTNISKOWIEC;
const k =  ?;
      N =  ?;

var na_lotniskowcu:  integer;
    DO.STARTU,  DO.LĄDOWANIA: condition;
    wolny: boolean;

procedure  zwolnij_pas;
begin
  if na_lotniskowcu  <  k then
    if not empty(DO_LADOWANIA) then
      signal(DO_LADOWANIA)
    else
      signal(DO.STARTU)
  else
    if not empty(DO_STARTU) then
      signal(DO.STARTU)
    else
      if  (na_lotniskowcu <  N) then
```

```
        signal(DO.LĄDOWANIA)
 end;
export procedure CHCE.STARTOWAĆ;
begin
  if not wolny then wait(DO_STARTU);
  wolny := false
end; {CHCĘ.STARTOWAĆ}

export procedure WYSTARTOWAŁ;
begin
  na_lotniskowcu := na_lotniskowcu -  1;
  wolny := true;
  zwolnij_pas
end; {WYSTARTOWAŁ}

export procedure CHCE.LĄDOWAĆ;
begin
  if not wolny or (na_lotniskowcu = N) then
   wait(DO_LĄDOWANIA);
  wolny := false
end; {CHCĘ.LĄDOWAĆ}

export procedure WYLĄDOWAŁ;
begin
  na_lotniskowcu := na_lotniskowcu + 1;
  wolny := true;
  zwolnij_pas
end; {WYLĄDOWAŁ}
begin
  na_lotniskowcu := 0;
  wolny := true
end;  {LOTNISKOWIEC}
```

4.4.9 Stolik dwuosobowy

Pierwsza osoba przybywająca na spotkanie musi poczekać na partnera. Do jej wstrzymania służy odpowiedni element tablicy CZEKA_NA_PARĘ. Druga osoba musi zająć kolejkę do stolika CZEKA_NA_STÓL. Gdy doczeka się już na stolik, zwalnia swojego partnera.

```
monitor KELNER;
const N = ?;
var CZEKA_NA_PARĘ: array [1..N] of condition;
    CZEKA_NA_STÓL: condition;
    przy_stole: integer;
 export procedure CHCĘ_STOLIK(j:  integer);
 begin
   if empty(CZEKA_NA_PARĘ[j])  then
     wait(CZEKA_NA_PARE[j])
   else begin
     if przy.stole  >  0  then  wait(CZEKA_NA_STÓL);
     przy_stole  :=  2;
     signal(CZEKA_NA_PARĘ[j])
   end
 end; {CHCĘ.STOLIK}

 export procedure ZWALNIAM;
 begin
   przy.stole  := przy_stole  -  1;
   if przy.stole =  0  then signal(CZEKA_NA_STÓL)
```

```
   end; {ZWALNIAM}

   begin
     przy_stole  := 0
   end;  {KELNER}
```

4.4.10 Zasoby dwóch typów

Procedura przydziału musi mieć parametr wejściowy określający, do której grupy należy proces zgłaszający żądanie. Procedura ta musi zwracać typ przydzielonego zasobom (ważne dla procesów grupy drugiej i trzeciej) oraz informację, czy zasób przydzielono (ważne dla procesów grupy trzeciej).

```
   monitor ZASOBY;
   const M =  ?;
         N =  ?;

   var wolneA, wolneB:.  integer;
       KOLEJKA1, KOLEJKA2:  condition;

   export procedure PRZYDZIEL(grupa: 1..3; var typ: A..B; var dostał: boolean);
   begin
      case grupa of
      l: begin
           if wolneA  =  0  then  wait(KOLEJKAl);
           wolneA  :=  wolneA -  l
         end;

  2: begin
       if (wolneA = 0) and  (wolneB = 0) then
       wait(KOLEJKA2);
       if wolneA > 0 then begin
         wolneA  := wolneA -  1;
         typ  := A
       end else
       if wolneB > 0 then begin
         wolneB  := wolneB -  1;
         typ  := B
       end
     end;
  3: if wolneB > 0 then begin
        wolneB  := wolneB -  1;
        dostał  := true
     end else
        if wolneA > 0 then  begin
        wolneA  := wolneA -  1;
        dostał  := true
     end else dostał  := false
  end
end; {PRZYDZIEL}

export procedure ZWOLNIJ(typ: A..B);
begin
  case typ of
  A: begin
      wolneA  := wolneA  + 1;
      if not  empty(KOLEJKAl) then signal(KOLEJKAl)
      else signal(KOLEJKA2)
```

```
    end;
  B: begin
       wolneB  := wolneB  + 1;
       signal(KOLEJKA2)
    end
  end
end; {ZWOLNIJ}

begin
  wolneA  := M;
  wolneB  := N
end; {ZASOBY}
```

Procesy pierwszej grupy mogą korzystać tylko z zasobów jednego typu, mają zatem wyższy priorytet niż pozostałe procesy. Dlatego po zwolnieniu zasobu typu A najpierw usiłuje się go przydzielić procesowi pierwszej grupy, a jeśli nie ma czekającego procesu z tej grupy, to procesowi drugiej grupy. Procesom trzeciej grupy najpierw próbuje się przydzielić gorszy zasób typu B, żeby zwiększyć szansę istnienia wolnego zasobu dla procesów grupy pierwszej.

4.4.11 Algorytm bankiera

Proces, którego żądanie nie może być w danej chwili obsłużone albo ze względu na brak jednostki zasobu, albo dlatego, że mogłoby to doprowadzić do stanu niebezpiecznego, musi czekać. W tym czasie mogą zgłosić się inne procesy, których żądania także z jakiś względów nie mogą być obsłużone. W chwili zwolnienia jednostki zasobu powstaje pytanie, któremu z czekających procesów ją przydzielić. Oczywiście przydziela się ją tylko wtedy, gdy nie doprowadzi to do stanu niebezpiecznego. Można więc, przeglądać kolejkę procesów czekających i przydzielić jednostkę zasobu pierwszemu, który zachowa stan bezpieczny. W ten sposób łatwo jednak doprowadzić do zagłodzenia procesów. Wystarczy, że procesy mające małe potrzeby będą intensywnie żądać zasobu, a wówczas proces, który ma duże niezaspokojone potrzeby, będzie czekał.

Warto w tym miejscu zauważyć, że we wszystkich znanych autorom książkach, w których jest opisany algorytm bankiera, problem zagłodzenia procesów nie jest w ogóle rozważany.

Wydaje się, że można uniknąć zagłodzenia obsługując procesy w takiej kolejności, w jakiej zgłosiły swe żądania. Inaczej mówiąc, pierwszy proces w kolejce czeka tak długo, aż zwolni się tyle jednostek zasobu, że przydzielenie mu jednej z nich nie doprowadzi do stanu niebezpiecznego. Jeśli zatem po zwolnieniu jednostki zasobu nie można jej przydzielić pierwszemu czekającemu procesowi, to nie przydziela się jej w ogóle. To rozwiązanie jest jednak jeszcze gorsze od poprzedniego, gdyż może prowadzić do blokady.

Brzmi to trochę dziwnie, bo przecież właśnie celem algorytmu bankiera jest unikanie blokady. Zakłada on jednak, że jeśli jakiś proces żąda jednostki zasobu i można ją przydzielić, to jest ona przydzielana. W powyższym rozwiązaniu tak nie jest i łatwo wyobrazić sobie sytuację, w której pierwszy czekający proces nigdy nie otrzyma jednostki zasobu, bo są one trzymane przez procesy czekające za nim.

Przedstawiamy tutaj rozwiązanie, w którym unika się zarówno zagłodzenia, jak i blokady. Pomysł polega na niedopuszczaniu do współzawodnictwa o jednostki zasobu procesów z zerowym kontem, jeśli nie zaspokojono wszystkich zgłoszonych wcześniej żądań. Procesy z zerowym kontem nie mają jednostek zasobu, a więc nie mogą blokować procesów, które mają już jakieś jednostki i żądają więcej. Ponieważ algorytm bankiera nie dopuszcza do

blokady, wszystkie żądania tych procesów będą w końcu zrealizowane i procesy z zerowym kontem będą mogły przystąpić do współzawodnictwa o jednostki zasobu. Należy jednak podkreślić, że unikanie zagłodzenia osiąga się tu kosztem dodatkowego zmniejszenia wykorzystania zasobu.

W kolejce NOWE umieszczamy procesy z zerowym kontem, w kolejce STARE te, które już coś mają. Z chwilą zwolnienia jednostki zasobu uaktywnia się wszystkie procesy z kolejki STARE, po czym każdy z nich sprawdza, czy pozostawi stan bezpieczny. Jeśli tak, to otrzymuje jednostkę zasobu, jeśli nie, to czeka dalej w kolejce STARE. Zmienna naboku wskazuje liczbę tych procesów, które zostały uaktywnione, ale jeszcze nie sprawdziły swojego warunku. Jeśli proces, który otrzymał jednostkę zasobu stwierdzi, że nie ma już procesów w kolejce STARE i nikt nie czeka na boku, to wpuszcza proces z zerowym kontem z kolejki NOWE. Proces ten natychmiast wpuszcza następny proces z tej kolejki, który wpuszcza następny. Tworzy się „kaskada" procesów, w wyniku której wszystkie aktualnie czekające procesy z kolejki NOWE będą dopuszczone do współzawodnictwa o zasól). Niektóre z nich mogą otrzymać jednostkę zasobu natychmiast, inne będą czekać w kolejce STARE.

```
monitor BANKIER;
const N =  ?;
var potrzeby,  przydział:  array[l..N]  of integer;
    dostępne:  integer;     {liczba  dostępnych  zasobów}
    naboku:  integer;
    NOWE,  STARE: condition;

function bezpieczny(i:  integer): boolean;

export procedure BIORĘ(i:  integer);
begin
  if not empty(STARE) and przydział [i]  = O then  begin
    wait(NOWE);
    signal(NOWE)          {kaskada  procesów}
  end;
  while not  bezpieczny(i) do begin
    wait(STARE);
    naboku := naboku +  l;
    signal(STARE);
    naboku := naboku -  l
  end;
  przydział [i]  :=  przydział[i] +  1;
  potrzeby [i]  := potrzeby [i] - 1;
  dostępne  := dostępne -  1;
  if (naboku = 0)  and empty(STARE) then signal(NOWE)
end; {BIORĘ}

export procedure ZWALNIAM(i:  integer);
begin
  dostępne  := dostępne +  1;
  przydział[i]  :=  przydział[i] - 1;
  potrzeby [i]  := potrzeby [i]  + 1;
  signal(STARE)
end; {ZWALNIAM}

begin
  dostępne  := M;
  naboku  := 0;
  for j  :=  l to N  do begin
    przydział[j]  := 0;
    potrzeby[j]  := f(j)
  end
```

```
 end;  {BANKIER}
```

4.4.12 Rodzina procesów

Stan każdego procesu jest opisany w elemencie tablicy opis. Indeks tego elementu jest jednocześnie numerem procesu. Opis zawiera numer procesu ojca oraz liczbę synów. Jeśli numer ojca jest równy O, to element jest wolny.

Dodatkowo każdy element zawiera zmienną typu condition, służącą do wstrzymywania procesu czekającego na zakończenie się jego synów. Opis procesu-antenata znajduje się w pierwszym elemencie tablicy.

```
 monitor PROCESY;
 const N =  ?;
 var i:  integer;
     ile:  integer;    {liczba procesów}
     opis:  array  [1..N] of record
                               ojciec:  integer;
                               ilusynów: integer;
                               CZEKA:  condition
                             end;
 export function NARODZINY(proces:  integer):  integer;
 var syn:  integer;
 begin
    if ile  =  N then
   NARODZINY  := O
 else begin
   syn := 2;
   while opis [syn].ojciec > O do
     syn := syn +  l;
   opis [syn].ojciec  := proces;
   opis [proces].ilusynów := opis [proces].ilusynów + 1;
   utwórz(syn);
   NARODZINY  := syn;
   ile := ile  + l
 end
end; {NARODZINY}

export procedure ŚMIERĆ(proces: integer);
var ojciec:  integer;
begin
 if proces  >  l then  begin
   if opis[proces].ilusynów > O  then
    wait(opis[proces].CZEKA);
   ojciec  :=  opis[proces].ojciec;
   opis[proces].ojciec := 0;
   opis[oj ciec].ilusynów := opis[ojciec].ilusynów - 1;
   if opis [oj ciec].ilusynów = O  then
     signal(oj ciec.CZEKA);
   zniszcz(proces);
   ile := ile - l
 end
end; {ŚMIERĆ}

begin
 for i :=  l to N  do  begin
   opis[i].ojciec  := 0;
   opis[i].ilusynów := O
 end;
```

```
   ile  :=  l
  end;  {PROCESY}
```

4.4.13 Szeregowanie żądań do dysku

Rozwiązanie wersji l

Ponieważ żądania są obsługiwane w kolejności ich zgłaszania, więc jeżeli dysk jest zajęty, to mogą być wstrzymywane w jednej kolejce.

```
monitor DYSK;
var  wolny: boolean;
     kolejka: condition;

export procedure  PRZYDZIEL(żądanie: par_żądania);
begin
   if not wolny  then wait(kolejka);
   wolny := false
end; {PRZYDZIEL}

export procedure  ZWOLNIJ;
begin
   wolny := true;
   signal(kolejka)
end; {ZWOLNIJ}
begin
   wolny := true
end; {DYSK}
```

Rozwiązanie wersji 2

Wydaje się, że dla implementacji strategii SCAN jest konieczne wstrzymywanie żądań do różnych cylindrów w odrębnych kolejkach, gdyż przy ich wznawianiu trzeba brać pod uwagę odległość żądanego cylindra od cylindra, nad którym aktualnie znajdują się głowice. Przeważnie jednak liczba żądań oczekujących jest niewielka, natomiast liczba cylindrów dosyć duża. Zatem takie rozwiązanie byłoby kosztowne.

Przedstawimy tu rozwiązanie, w którym używamy niestandardowego mechanizmu priorytetowego wstrzymywania procesów. Implementacja tego mechanizmu jest tylko niewiele trudniejsza niż zwykłego wstrzymywania procesów.

Istota rozwiązania polega na zauważeniu, że do wyboru żądania do wznawiania nie jest potrzebna informacja, cło których cylindrów żądania są skierowane. Musimy jedynie wiedzieć, które żądanie znajduje się najbliżej głowic, przy czym trzeba odróżnić żądania do cylindrów, w kierunku których przesuwają się głowice, od żądań do pozostałych cylindrów. Potrzebne są zatem dwie kolejki żądań, przed i za głowicami, uporządkowane l według odległości od głowic. Głowice jednak stale zmieniają położenie, a i więc zmieniają się też odległości. Trzeba wobec tego ustalać odległość żądań od jakichś stałych punktów odniesienia. Takimi punktami mogą być skrajne cylindry. Uporządkowanie według malejącej kolejności od skrajnego cylindra j jest takie samo, jak uporządkowanie według rosnącej odległości od głowic.

Niech lewa_kolejka dotyczy żądań skierowanych do cylindrów o numerach mniejszych niż numer cylindra bieżącego, czyli tego, nad którym znajdowały się głowice podczas wykonywania ostatniego żądania. Natomiast prawa_kolejka dotyczy żądań skierowanych do cylindrów o numerach większych niż numer cylindra bieżącego.

Żądania w lewej kolejce są uporządkowane malejąco według numerów cylindrów, natomiast w prawej kolejce — rosnąco. Zatem miejsce żądania przy wstawianiu do prawej kolejki jest określone po prostu numerem żądanego cylindra. Jeżeli chcemy zastosować ten sam mechanizm do wstawiania żądań do obu kolejek, to w przypadku żądań wstawianych do lewej kolejki właściwą kolejność osiągniemy podając różnicę liczby cylindrów dysku i numeru żądanego cylindra.

Wstawianie procesu do kolejki k (wstrzymywanie procesów) w miejsce określone liczbą całkowitą p realizuje operacja wait(k.p). Procesy w kolejce są uporządkowane ze względu na rosnące wartości liczb, podawanych przy operacji wstrzymywania. Zatem liczby te możemy interpretować jako priorytet procesu przy operacji wznowienia. Jeżeli liczba jest taka sama dla wielu procesów, to o kolejności procesów decyduje kolejność wstrzymania, tak jak przy standardowej operacji wait.

Prezentowane rozwiązanie, którego autorem jest Hoare [Hoar74] zaczerpnięto z książki [Ma0087].

```
monitor DYSK;
const  C = ?;
var  wolny: boolean;
    cylinder:  l..C;
    kierunek:  (lewy,  prawy);
    lewa.kolejka, prawa_kolejka: condition;

export procedure  PRZYDZIEL(żądanie: par_żądania);
 begin
   if not wolny then
     if żądanie.cylinder <  cylinder or
       żądanie.cylinder = cylinder and  kierunek =  lewy
     then wait(lewa_kolejka,  C-żądanie.cylinder)
     else wait(prawa_kolejka,  żądanie.cylinder);
   cylinder  :=  żądanie.cylinder;
   wolny  := false
  end; {PRZYDZIEL}

export procedure  ZWOLNI J.
begin
  wolny  := true;
  if kierunek  = lewy then  begin
  if  empty(lewa_kolejka)
  then begin
    kierunek  := prawy;
    signal(prawa_kolejka)
  end
  else signal(lewa_kolejka)
end else begin
  if  empty(prawa_kolejka)
  then begin
    kierunek  := lewy;
    signal(lewa_kolejka)
  end
  else signal(prawa_kolejka)
end
end; {ZWOLNIJ}
begin
  wolny := true;
  kierunek := lewy;
  cylinder := C div  2
end; {DYSK}
```

Warto zwrócić uwagę na pewne subtelne szczegóły tego rozwiązania. Jeżeli dysk jest zajęty, kierunek głowicy jest lewy i przybywa żądanie skierowane do bieżącego cylindra, to jest ono umieszczane w kolejce lewa_kolejka (analogicznie, jeżeli kierunek jest prawy, to żądanie jest umieszczane w kolejce prawa_kolejka). Po obsłudze-bieżącego żądania będzie wznowione kolejne żądanie właśnie z tej kolejki i kierunek ruchu głowic nie zmieni się. Jeżeli zatem intensywnie będą zgłaszane żądania do tego samego cylindra, to zagłodzą one żądania oczekujące do innych cylindrów. Aby tego uniknąć, wystarczy zamienić instrukcje wait. Wtedy żądanie skierowane do bieżącego cylindra będzie wstawione do kolejki prawa_kolejka, gdy kierunek jest lewy, a do kolejki lewa_kolejka, gdy kierunek jest prawy. Wobec tego żądania te zostaną wykonane przy następnym przejściu głowic nad tym cylindrem (po zmianie kierunku ruchu głowic). Odnosi się to jednak tylko do tych żądań do bieżącego cylindra, które przybyły podczas wykonywania żądania dotyczącego tego cylindra.

Drugi problem dotyczy zmiany kierunku ruchu głowic. Zmiana ma miejsce tylko wtedy, gdy po zakończeniu wykonywania żądania nie ma już dalszych żądań przed głowicami, natomiast są żądania za nimi. Jeżeli jednak zakończy się ostatnie z oczekujących żądań, to dysk staje się po prostu wolny i na zmiennej kierunek będzie zachowany kierunek ostatnio wykonywanego ruchu głowic. Nowe żądanie zastaje wolny dysk i może być bezzwłocznie wykonane. Jednak wykonanie tego żądania może wymagać ruchu głowic w kierunku przeciwnym niż wskazywany przez zmienną kierunek. W pewnych sytuacjach może to spowodować niepotrzebne zwiększenie średniego czasu wykonania żądań, a poza tym program jest niezgodny ze specyfikacją.

Aby usunąć błąd, wystarczy ustalać kierunek ruchu głowic każdorazowo przy rozpoczynaniu wykonywania kolejnego żądania.

Następujące rozwiązanie jest pozbawione obu opisanych błędów. Pierwszy z nich został usunięty już wcześniej (np. w [Holt83]), drugi natomiast jest konsekwentnie powtarzany we wszystkich znanych autorom książkach i artykułach.

```
monitor DYSK;
const  C =  ?;
var wolny:  boolean;
   cylinder:  l..C;
   kierunek:  (lewy, prawy);
   lewa_kolejka,  prawa_kolejka:  condition;

export procedure PRZYDZIEL(żądanie:  par_żądania);
begin
  if not wolny then
   if żądanie.cylinder  >  cylinder or
      żądanie.cylinder =  cylinder and kierunek = lewy
   then wait(prawa_kolejka, żądanie.cylinder)
   else wait(lewa_kolejka, C-żądanie.cylinder);
  if żądanie.cylinder <  cylinder then
   kierunek := lewy
  else
  if żądanie.cylinder >  cylinder then
   kierunek := prawy;
  cylinder := żądanie.cylinder;
  wolny  := false
end; {PRZYDZIEL}

export procedure ZWOLNIJ;
begin
  wolny  := true;
  if kierunek = lewy then begin
    if empty(lewa_kolejka)
```

```
      then  signal(prawa_kolejka)
       else  signal(lewa_kolejka)
     end  else begin
       if empty(prawa_kolejka)
       then  signal(lewa_kolejka)
       else  signal(prawa_kolejka)
     end
   end; {ZWOLNIJ)
   begin
     wolny  :=  true;
     cylinder  := C div 2
   end; {DYSK}
```

4.4.14 Asynchroniczne wejście-wyjście

Jeżeli proces żąda wejścia-wyjścia, gdy dysk jest zajęty, to żądanie zapamiętuje się w kolejce żądań do_wykonania. Przebywa tam tak długo, aż zostaną wykonane żądania wcześniejsze. Żądanie usuwa się z tej kolejki w procedurze PRZERWANIE_WE_WY.

Proces wywołujący procedurę CZEKAM_NA_WYKONANIE albo jest wstrzymywany, gdy żądanie jeszcze nie jest wykonane, albo nie, gdy żądanie już zostało wykonane. Żądania wykonane znajdują się w kolejce po_wykonaniu.

Są tam wstawiane w procedurze PRZERWANIE_WE_WY, jeżeli proces, który wygenerował żądanie nie czeka na nie. Sprawdzenie tego faktu odbywa się po ewentualnej inicjacji wykonania kolejnego żądania, — chodzi o to, aby dysk nie był niepotrzebnie bezczynny.

W rozwiązaniu używa się funkcji i procedur działających na kolejkach żądań. Treści tych funkcji i procedur pomijamy.

```
   monitor  DYSK;
   type kolejka_żądań = ...;
   var KOLEJKA:  condition;
       wolny,  czeka: boolean;
       bieżące,  wykonane: par_żądania;
       do_wykonania, po.wykonaniu: kolejka_żądań;

   function  pusta(k: kolejka_żądań):  boolean;
      {true, gdy kolejka k jest pusta; wpp false}
   function  w_kolejce(ż: par.żądania;
                       k: kolejka.żądań):  boolean;
      {true, gdy żądanie ż jest w kolejce k;  wpp false}
   procedure z_kolejki(ż: param.żądań; k:  kolejka_żądań);
      {usuwa żądanie ż z kolejki k}
   procedure do_kolejki(ż: param.żądań;  k:  kolejka_żądań);
      {wstawia żądanie ż do kolejki k}
   function  pierwsze(k: kolejka.żądań):  par_żądania;
      {dostarcza pierwsze żądanie z kolejki k, usuwając je}
export procedure ŻĄDANIE_WE_WY(żądanie:  par.żądania);
begin
  if  wolny then begin
   wolny := false;
   bieżące := żądanie;
   start_dysk(bieżące)
  end
  else
   do_kolejki(żądanie,  do.wykonania)
end;  {ŻĄDANIE_WE_WY>
```

```
export procedure CZEKAM_NA_WYKONANIE(żądanie:par_żądania);
begin
  if  w_kolejce(żądanie,  po_wykonaniu) then
     z.kolejki(żądanie,  po.wykonaniu)
  else
  repeat
   wait(KOLEJKA)
  until żądanie = wykonane;
  czeka := true
end;  {CZEKAM_NA_WYKONANIE}

export procedure PRZERWANIE_WE_WY;
begin
  wykonane := bieżące;
  if  pusta(do_wykonania) then
   wolny := true
  else begin
   bieżące := pierwsze(do_wykonania);
    start_dysk(bieżące)
  end;
  czeka := false;
  while not empty(KOLEJKA) and not czeka do
    signal(KOLEJKA);
  if  not czeka then
   do_kolejki(wykonane, po_wykonaniu)
end;  {PRZERWANIE.WE.WY}

begin
  wolny := true
end;  {DYSK}
```

4.4.15 Dyskowa pamięć podręczna

W rozwiązaniu zrealizowano strategię LRU za pomocą stosu. Ramki pamięci podręcznej powiązano w dwukierunkową listę, której końce są wskazywane przez zmienne początek i koniec. Gdy sektor zostaje wczytany do pamięci operacyjnej, procedura na_początek przemieszcza go na początek listy. Zatem na końcu listy będą się znajdowały sektory najdawniej używane.

Każda ramka jest reprezentowana przez rekord zawierający pole adres, umożliwiające identyfikację znajdującego się w ramce sektora. Pola poprz i nast służą do organizacji listy. W kolejce KONIEC_CZYTANIA są wstrzymywane procesy oczekujące na wczytanie sektora, a w polu czytany odnotowuje się, że do danej ramki jest wczytywany sektor.

Na początku procedury ŻĄDANIE_ODCZYTU następuje wyszukanie żądanego sektora w pamięci podręcznej. Realizuje to funkcja znajdź. Jeśli żądany sektor jest w pamięci podręcznej (PP[znajdź(żądanie)] .adres = żądanie), to jej wartością jest numer ramki, a w przeciwnym razie 0. Jeżeli sektora nie ma w pamięci podręcznej, a zostało zainicjowane wczytywanie sektorów do wszystkich ramek (zmienna ile_czytanych ma wartość R), to żądanie jest wstrzymywane. Do wstrzymywania takich żądań służy kolejka WSZYSTKIE_CZYTANE. Jeśli natomiast jest jakaś ramka, którą można użyć, to poszukuje się jej w pętli repeat, poczynając od końca listy, czyli od sektorów, które były najdawniej używane. Podczas przeszukiwania są pomijane ramki, do których zainicjowano wczytywanie sektorów (pole czytany). Po znalezieniu ramki, którą można wykorzystać, do pola adres są wpisywane parametry żądania, które będą już od tej chwili stosowane do identyfikacji sektora

wczytywanego do tej ramki. Kolejne żądania wczytania, tego samego sektora będą wstrzymywane na zmiennej KONIEC_CZYTANIA, tym samym co pierwsze żądanie.

W procedurze KONIEC_ODCZYTU jest potrzebny numer ramki, do której zakończyła się właśnie transmisja sektora. Numer ten jest wyznaczany na podstawie parametrów żądania, które są przekazywane z procedury obsługi przerwania wejścia-wyjścia, wywołującej procedurę KONIEC_ODCZYTU.

Podczas inicjacji struktur danych monitora w polu adres każdej ramki zapisuje się parametry nieistniejących sektorów, aby funkcja znajdź zwracała wartość 0.

```
   monitor PAMIĘĆ.PODRĘCZNA;
   const R = ?;
         C = ?;

   var i,   ile_czytanych:    integer;
        początek, koniec:    integer;
        WSZYSTKIE.CZYTANE:  condition;
    PP:  array[l..R]  of  record
                            adres:  par_żądania;
                            poprz,  nast:  integer;
                            KONIEC.CZYTANIA:  condition;
                            czytany:  boolean
                          end;

procedure  na_początek(ramka:  integer);  {przemieszcza  wskazaną  ramkę  na  początek
listy LRU}

function znajdź(żądanie:  par_żądania):  integer;
  {jeśli żądany sektor  jest  w pamięci  podręcznej,
   tzn.  PP[znajdź(żądanie)].adres = żądanie,  to
   zwraca numer tej  ramki, wpp zwraca  0}

export procedure ŻĄDANIE_ODCZYTU(żądanie:  par.żądania;
                                 adresPO:  integer);
var ramka: integer;
    jest: boolean;
begin
  ramka := znajdź(żądanie);
  if ramka = O then begin
    if ile.czytanych =  R then wait(WSZYSTKIE_CZYTANE);
    ramka := koniec;
    jest := false;
    repeat
      if PP[ramka].czytany then
        ramka := PP[ramka].poprz
      else jest :=  true
    until jest;
    PP[ramka].adres   := żądanie;
    PP[ramka].czytany := true;
    ile.czytanych := ile.czytanych + 1;
    DYSK.ŻĄDANIE.WE.WY(żądanie);
    wait(PP[ramka].KONIEC.CZYTANIA)
  end
  else
    if PP[ramka].czytany then
      wait(PP[ramka].KONIEC.CZYTANIA);
  na.początek(ramka);
  zPPdoPO(ramka, adresPO)
end; {ŻĄDANIE.ODCZYTU}
   export procedure KONIEC_ODCZYTU(żądanie: par_żądania);
   var ramka:  integer;
```

```
begin
  ramka  :=  znajdź(żądanie);
  repeat
    signal(PP[ramka].KONIEC.CZYTANIA)
  until  empty(PP[ramka].KONIEC.CZYTANIA);
  PP[ramka].czytany := false;
  ile_czytanych := ile.czytanych  -  1;
  if ile.czytanych = R - l then signal(WSZYSTKIE_CZYTANE)
end; {KONIEC.ODCZYTU}

begin
  początek  := 1;
  koniec    := R;
  for i  :=  l  to R do
  with PP[i]  do begin
    poprz  :=  i - 1;         {O =  nil}
    nast   :=  i + 1;
    czytany := false;
    adres.cylinder  := C +  l
  end;
  PP[R].nast  := 0;          {O =  nil}
  ile_czytanych := O
end;  <PAMIĘĆ_PODRĘCZNA}
```

Ponieważ procedura zPPdoPO jest wykonywana synchronicznie, więc w tym czasie dostęp do monitora jest zablokowany.

4.4.16 Pamięć operacyjna — strefy dynamiczne

Założenia o wstrzymywaniu żądań oraz o kolejności ich wznawiania są bardzo istotne dla tego zadania. W przypadku przydziału dynamicznego pamięć operacyjna szybko ulega fragmentacji i żądania dużych ilości pamięci nie mogłyby być zrealizowane — zostałaby zagłodzone. Warto jednak zauważyć, że taki sposób realizacji żądań powoduje zmniejszenie wykorzystania pamięci.
Aby wyróżnić pierwsze wstrzymane żądanie wśród wszystkich wstrzymanych, wprowadzimy dwie kolejki: PIERWSZE i NASTĘPNE. Żądanie pierwsze będzie wznawiane za każdym razem, gdy zwolni się pamięć. Jeżeli nic będzie można go zrealizować, to będzie znów wstrzymywane. Wznowione żądanie po otrzymaniu pamięci będzie wznawiało kolejne wstrzymane żądanie.
Każdą zajętą strefę opisuje rekord typu strefa, w którym są przechowywane: adres początku i adres końca strefy oraz wskaźniki do rekordów opisujących sąsiednie strefy. Rekordy są powiązane w listę dwukierunkową, uporządkowaną rosnąco według adresów stref. Początek listy znajduje się w zmiennej początek. Na początku i na końcu listy znajdują się dwie fikcyjne strefy, rozpoczynające się odpowiednio od adresów O i rozmiar, obie zerowej wielkości. Ich istnienie ułatwia modyfikowanie listy stref.

```
monitor PAMIĘĆ.OPERACYJNA;
const rozmiar = ?;
type strefa = record
                pocz,  koń:  integer;
                poprz,  nast:  ^strefa
              end;
var PIERWSZE, NASTĘPNE:  condition;
    początek: ^strefa;
```

```
      chce: integer;

   export procedure PRZYDZIEL(var adres:integer;  ile:integer);
     function znaleziona(var adres: integer;
                         ile:  integer): boolean;
     var s, t, u: ^strefa;
         jest: boolean;
     begin
       s := początek;
       t := początek^.nast;
       jest := false;
       repeat
         if t^.pocz-(s~.kon+1)  >= ile then jest := true
         else begin
            s  := s^.nast;
            t  := t^.nast
         end
       until jest or  (t =  nil);
       if jest then begin
         new(u);
         u^.pocz := s^.koń + 1;
         u^.koń  := u^.pocz +  ile - 1;
         u^.poprz  :=  s;
         u^.nast   :=  t;
         t^.poprz  :=  u;
         s^.nast   :=  u;
         adres  := u^.pocz
    end;
    znaleziona := jest
  end; -[znaleziona}
begin {PRZYDZIEL}
  if not empty(PIERWSZE) then wait(NASTĘPNE);
  while not znaleziona(adres, ile) do begin
    chce := ile;
    wait(PIERWSZE);
  end;
  signal(NASTĘPNE)
end; {PRZYDZIEL}

export procedure ZWOLNIJ(adres:  integer);
var s, t, u: ^strefa;
begin
  u := początek;
  while u^.pocz <> adres do  u := u^.nast;
  s := u^.poprz;
  t := u^.nast;
  s^.nast := t;
  t^.poprz  := s;
  dispose(u);
  if not empty(PIERWSZE) then
  if t^.pocz - (s^.koń + 1) >= chce then signal(PIERWSZE)
end; {ZWOLNIJ}

begin
  new(początek);
  new(s);
  with początek^ do begin
    pocz  := 0;
    koń   := 0;
    poprz := nil;
    nast  := s
```

```
  end;
  with s^ do begin
    pocz  := rozmiar;
    koń   := rozmiar;
    poprz := początek;
    nast  := nil
  end;
end;  {PAMIĘĆ.OPERACYJNA}
```

Funkcja znaleziona wyszukuje zgodnie ze strategią. First Fit wolną strefę pamięci o rozmiarze nie mniejszym od żądanego. Jeśli znajdzie się taką strefę, to tworzy się nową zajętą i wstawia ją do listy stref. Jeżeli się nie znajdzie, to zapotrzebowanie procesu zapamiętuje się w zmiennej chce i proces jest wstrzymywany.

Pętla while w procedurze PRZYDZIEL jest potrzebna tylko po to, aby proces wstrzymany na warunku PIERWSZE wykonał po wznowieniu funkcję znaleziona. Wartością tej funkcji na pewno będzie true (w procedurze ZWOLNIJ sprawdzono, że jest dostatecznie duża strefa), ale nastąpi w niej przydział strefy wznawianemu procesowi.

Procedura ZWOLNIJ wyszukuje strefę i usuwa ją z listy zajętych. Jeżeli jest jakiś wstrzymany proces i jego zapotrzebowanie można zaspokoić, to ten proces będzie wznowiony.

4.4.17 Strategia buddy

Monitor zarządzający listą wolnych obszarów o wielkości c * 21 będzie zawierał kolejkę KOLEJKA procesów czekających na wolny obszar. Liczbę wolnych obszarów będzie określać zmienna wolne. W zmiennej czeka będzie się pamiętać liczbę czekających procesów, a w naile — liczbę wolnych obszarów, na które te procesy czekają. Pierwszy proces, który zaczyna czekać, czeka bowiem nie na jeden wolny obszar, lecz na dwa powstałe w wyniku podziału obszaru większego. Gdy zaczyna czekać drugi proces, oba nadal czekają tylko na dwa wolne obszary. Gdy pojawi się trzeci proces, liczba obszarów, na które będą teraz czekały, wyniesie cztery itd. Ogólnie czeka się zawsze na parzystą liczbę wolnych obszarów, ale liczba procesów czekających może być nieparzysta.

Proces, który żąda wolnego obszaru w chwili, gdy ich nie ma, zwiększa liczbę czekających, a następnie sprawdza, czy nie przekroczył w ten sposób liczby oczekiwanych obszarów. Jeśli przekroczył, to inicjuje proces DZIEL wywołując procedurę start.dziel.

Po wznowieniu z kolejki KOLEJKA proces otrzymuje adres żądanego obszaru w zmiennej taca. Dzięki jej zastosowaniu, zaoszczędzono zbędnego wstawiania do listy wolnego obszaru, który i tak za chwilę będzie z niej usunięty.

W przypadku, gdy nikt nie czeka na obszar, w procedurze ZWOLNIJ trzeba wyznaczyć adres kumpla i ewentualnie odszukać go na liście. W tym celu określa się, którą połówkę proces zwalnia. Jeśli adres zwalnianego obszaru dzieli się całkowicie przez c * 2+i, to jest to pierwsza połówka, trzeba więc szukać drugiej.

W procedurze KONIEC_DZIEL trzeba uwzględnić trzy przypadki. Jeśli już nikt nie czeka na wolny obszar (bo czekające procesy otrzymały je w wyniku wykonania procedury ZWOLNIJ), to oba obszary trzeba ponownie połączyć. Jeśli jakiś proces czeka, to należy mu podać wolny obszar w zmiennej taca. Gdy sterowanie wróci do procedury KONIEC_DZIEL sprawdza się znowu, czy ktoś czeka. Jeśli tak, to należy mu podać „na tacy" adres drugiego wolnego obszaru, jeśli nie, to wolny obszar trzeba wstawić do listy wolnych obszarów.

```
  const K =  ?;
```

```
  monitor BUDDY(k: 0..K);
  var KOLEJKA:  condition;
    wolne: integer;          {liczba wolnych obszarów}
    czeka: integer;          {liczba czekających procesów}
    naile: integer:          {liczba oczekiwanych obszarów}
    taca:  adres;            {do podawania obszarów
                               czekającym procesom}
  export procedure PRZYDZIEL(var a: adres);
  begin
    if wolne =  O then begin
      czeka  :=  czeka +  1;   {będzie czekać}
      if czeka  > naile  then begin
        naile  := naile  + 2; {potrzebny  nowy podział}
        start_dziel(k + 1)
      end;
      wait(KOLEJKA);         {czeka  na obszar}
      a  := taca             {dostaje  na tacy}
    end
    else begin               {są wolne obszary}
      usuń(k,  a);            {usuwa  jeden z nich}
      wolne  :=  wolne -  l   {już nie  jest wolny}
    end
  end; {PRZYDZIEL}

  export procedure  ZWOLNIJ(a:  adres);
  var d:  integer;            {kierunek szukania}
  begin
    if czeka >  O then begin {ktoś  czekał}
      czeka  :=  czeka -  1;   {przestaje czekać}
      taca  :=  a;             {dostanie obszar na tacy}
      signal(KOLEJKA)      {przestanie czekać}
    end
    else begin             {jest  wolny obszar}
      if  a mod c  *  2*(k +  1)  = O
      then d := l           {który  kumpel}
  else d  :=  -1;
  if znajdź_usuń(k,  a  +  d  *  c  *  2~k)  then
  begin                    {znalazł się  kumpel}
    wolne  := wolne  - 1;  {już go  nie ma na liście}
    start_łącz(k  +l,a, a+d*c*  2~k)
  end
  else begin            {nie ma  kumpla i nikt nie czeka}
    wstaw(k, a);
    wolne  := wolne  + l  {jeden wolny  więcej}
  end
 end
end; {ZWOLNIJ}

export procedure  KONIEC_DZIEL(al,a2:  adres);
begin
  naile  := naile  -  2;
  if czeka > O then begin
   czeka  := czeka  -  l;
   taca  :=  al;
   signal(KOLEJKA);
   if czeka > O  then begin
     czeka  := czeka  - 1;
     taca  :=  a2;
     signal(KOLEJKA)
   end
   else begin
```

```
    wstaw(k, a2);
    wolne  := wolne  + l
   end
  end
  else start_łącz(k + l, al, a2)
end; {KONIEC.DZIEL}

begin
 czeka  :=  0;
 naile  :=  0;
 if k = K  then begin
  wolne  :=  1;
  wstaw(K, 0)
 end
 else wolne  := O
end;   {BUDDY}
```

Treści procesów DZIEL i ŁĄCZ są bardzo proste. Proces DZIEL (k) musi przydzielić obszar o wielkości c * 2k a następnie wywołać KONIEC_DZIEL w monitorze BUDDY(k-l). Proces LĄCZ(k,al ,a2) musi zwolnić obszar o adresie, który jest mniejszą z liczb al i a2, wywołując procedurę ZWOLNIJ w monitorze BUDDY (k).

```
  process  DZIEL(k: 1..K);
  var a:  adres;
  begin
    BUDDY(k).PRZYDZIEL(a);
    BUDDY(k  -  l).KONIEC_DZIEL(a,  a + c * T (k  -  D)
  end;  {DZIEL}

  process  LĄCZ(k: 1..K; al, a2:  adres);
  begin
    BUDDY(k).ZWOLNIJ(min(al,a2))
  end;  {ŁĄCZ}
```

5. Symetryczne spotkania

5.1 Wprowadzenie

5.1.1 Trochę historii

W roku 1978 C. A. R. Hoare opublikował artykul pt. „Communicating Seąuential Processes" [HoarTS], w którym zaproponował nowy język do zapisywania procesów współbieżnych. Został on nazwany CSP od pierwszych liter tytułu tego artykułu. Dzięki swej prostocie i przejrzystości, a zarazem sile wyrazu język CSP zrobił olbrzymią karierę. Powstały liczne prace rozważające jego teoretyczne aspekty oraz wiele jego realizacji.

W swej pierwotnej formie CSP zrealizowane jako CSP-80. Zbliżoną realizacją jest occam przeznaczony przede wszystkim do programowania trausputerów. Twórcy języka Ada zapożyczyli z CSP ideę spotkań, trochę ją jednak modyfikując.

W roku 1985 C. A. R. Hoare wydał książkę [Hoar85], w której szczegółowo i metodycznie przedstawił ideę komunikujących się procesów sekwencyjnych, a także podał podstawy matematyczne opisywanych mechanizmów. W stosunku do pierwotnego artykułu istotnie zmieniła się notacja. Książkowe CSP jest bardziej formalizmem służącym do specyfikacji niż językiem programowania. Od roku 1978 zmieniły się także niektóre koncepcje. Hoare zrezygnował m.in. z jawnego wskazywania procesów przy komunikacji i wprowadził komunikację przez kanały, zmienił również zasady kończenia się procesów.

Z dwóch powodów w niniejszym rozdziale oprzemy się na notacji zaproponowanej we wcześniejszej pracy Hoare'a. Po pierwsze, jest to notacja języka imperatywnego, a takimi językami są wszystkie inne języki, do których odwołujemy się w tej książce. Po drugie, brak kanałów ma pewną zaletę. Oprócz tego, że nie trzeba tworzyć nowych bytów, odpada problem sprawdzania, czy z kanału korzystają dokładnie dwa procesy i czy jeden z nich jest nadawcą a drugi odbiorcą. Upraszcza to zapis procesów, a tym samym zrozumienie, jak one działają. Wadą jest to, że procesy muszą znać nawzajem swoje nazwy, co utrudnia tworzenie bibliotek.

Dlatego języki programowania oparte na CSP, takie jak occam, Parallel C czy Edip, wymagają komunikacji przez kanały. Języki te omówimy dokładniej na końcu rozdziału.

Nie będziemy przedstawiać całego języka CSP. Pominiemy szczegóły, które nie są konieczne do zrozumienia przykładów i rozwiązania podanych zadań, a zwłaszcza nie będziemy omawiać sposobu komunikacji programu ze światem zewnętrznym.

5.1.2 Struktura programu

Program w CSP składa się z ujętego w nawiasy kwadratowe ciągu treści procesów oddzielonych od siebie znakiem „||", oznaczającym, że procesy będą wykonywane równolegle. Treść każdego procesu jest poprzedzona etykietą, będącą jednocześnie nazwą procesu, po której następują dwa dwukropki.

Ogólny schemat programu wygląda więc następująco:

```
Pl::<treść Pl>  || P2::<treść P2>  ||  ...  ||  Pn::<treść Pn>
```

Procesy możemy parametryzować tworząc tablicę procesów. Tak więc

```
P(k:l..N)::  <treść zależna od zmiennej związanej k>
```

jest skrótowym zapisem N procesów o nazwach odpowiednio P(l) , ...,P(N), przy czym treść procesu P (i) powstaje przez zastąpienie zmiennej związanej k stałą i. Możliwe jest tworzenie wielowymiarowych tablic procesów w naturalny sposób.

5.1.3 Instrukcje

Charakterystyczną cechą języka CSP jest prostota i zwięzłość. Wyróżnia się w nim cztery rodzaje instrukcji prostych: przypisania, pustą, wejścia i wyjścia oraz tylko dwa rodzaje instrukcji strukturalnych: alternatywy i pętli. (Instrukcje wejścia i wyjścia pełnią w CSP szczególną rolę i dlatego omówimy je oddzielnie w punkcie 5.1.4) Oprócz sekwencyjnego wykonania instrukcji jest możliwe także wykonanie równolegle w obrębie jednego procesu.

Przypisanie

Oprócz zwykłej instrukcji przypisania, w CSP jest możliwe także przypisanie jednoczesne. Wówczas po lewej stronie znaku „:=" zamiast nazwy zmiennej znajduje się ujęty w nawiasy ciąg nazw zmiennych oddzielonych przecinkami, a po prawej stronie znajduje się odpowiadający mu ciąg wyrażeń, także oddzielonych przecinkami i ujętych w nawiasy. Wykonanie takiej instrukcji polega na równoległym obliczeniu wartości wyrażeń, a następnie jednoczesnym przypisaniu ich odpowiadającym im zmiennym. Przykładowo,

```
(x,y)  :=  (y,x)
```

jest jednoczesną zamianą wartości x i y. Dla uproszczenia przyjmujemy, że wyrażenia można tworzyć za pomocą takich samych operatorów jak w Pascalu.

Instrukcja pusta

Instrukcję pustą oznacza się słowem kluczowym skip. Potrzebę jej wprowadzenia wyjaśniamy w następnym punkcie.

Instrukcja alternatywy

W CSP, zamiast zwykłej instrukcji warunkowej, jest dostępna ogólniejsza instrukcja alternatywy wzorowana na instrukcjach dozorowanych Dijkstry (por. [Dijk78]). Instrukcję alternatywy zapisuje się według schematu

```
[Dl ->  II  []  D2  -> 12 []  ...   []  Dn  ->  In]
```

Dozór Di jest niepustym ciągiem warunków logicznych oddzielonych średnikami. Średnik w tym przypadku oznacza operator logiczny and. Warunki w dozorze są obliczane po kolei (począwszy od lewej strony), a po napotkaniu pierwszego fałszywego warunku obliczanie jest przerywane. Symbol [] oznacza tu niepusty ciąg instrukcji oddzielonych średnikami. Średnik w tym przypadku jest operatorem następstwa. (Między warunkami w dozorze i między instrukcjami mogą się pojawiać także deklaracje zmiennych, por. 5.1.5.)

Instrukcje dozorowane w obrębie instrukcji alternatywy można parametryzować w następujący sposób:

```
(k:l..N)  dozór  zależny od k ->  instrukcje  zależne od  k
```

Taki zapis jest równoważny wypisaniu N odpowiednich dozorów i N następujących po nich ciągów instrukcji, w których w miejsce zmiennej związanej k jest wstawiony numer kolejny dozoru.

Wykonanie instrukcji alternatywy polega na równoczesnym obliczeniu wszystkich jej dozorów, niedeterministycznym wybraniu jednego dozoru spośród tych, które są spełnione i wykonaniu następującego po tym dozorze ciągu instrukcji.

Jeśli w instrukcji alternatywy żaden dozór nie jest spełniony, uważa się to za błąd powodujący przerwanie wykonania programu.

Tak więc instrukcja [x>0 -> x:=l] spowoduje błąd, gdy x < O (jeśli zawsze x > O, nie ma sensu używania instrukcji alternatywy). Efekt instrukcji pascalowej postaci

```
if w  then I można uzyskać w CSP pisząc
    [ w  -> I  [] not  w -> skip]
```

Pętla

Instrukcję pętli zapisuje się dostawiając gwiazdkę przed instrukcją alternatywy. Pętla w CSP ma więc następującą postać:

```
*[D1 -> II  []  D2 ->  12  []  ...   []  Dn -> In]
```

Wykonuje się ją tak długo, aż nie będzie spełniony żaden z jej dozorów. Jeśli w danej chwili jest spełniony więcej niż jeden dozór, podobnie jak w instrukcji alternatywy, niedeterministycznie jest wybierany jeden z nich i wykonuje się następującą po nim instrukcję dozorowaną.

Wykonanie równolegle

Równolegle wykonanie kilku ciągów instrukcji zapisuje się w CSP następująco

```
I1 || I2 || ... || In
```

Komentarze

Komentarze w CSP poprzedzamy słowem kluczowym comment i kończymy średnikiem. (Warto w tym miejscu zauważyć, że w CSP występują tylko dwa słowa kluczowe: comment i skip.)

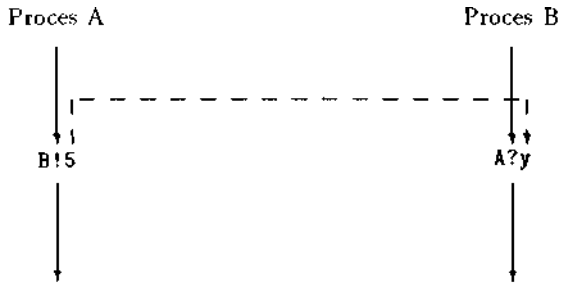
5.1.4 Spotkania

W CSP procesy komunikują się ze sobą w sposób synchroniczny za pomocą instrukcji wejścia-wyjścia. Instrukcja wyjścia postaci B!X(yl, ... ,yn) w procesie A oznacza, że proces A chce wysłać do procesu B ciąg wartości wyrażeń y1, ..., yn, identyfikowany nazwą X. Aby było to możliwe, w procesie B musi znajdować się dualna instrukcja wejścia A?X(z1, . . . ,zn), przy czym z1, . . . ,zn są zmiennymi odpowiednio tego samego typu co wyrażenia y1, ..., yn.

Instrukcja wejścia-wyjścia jest wykonywana wtedy, gdy sterowanie w procesie A osiągnie instrukcję wyjścia, a sterowanie w procesie B odpowiadającą jej instrukcję wejścia (zwykle więc jeden z procesów musi czekać na drugi). Wówczas odbywa się spotkanie, które polega na jednoczesnym wykonaniu ciągu przypisań zi := yi, dla i = 1,...,n. Parę instrukcji wejścia i wyjścia można więc uważać za dwie części tej samej instrukcji wejścia-wyjścia, a jej wykonanie za „rozproszone przypisanie". Ideę spotkania ilustruje rys. 5.1.

Ciąg wyrażeń w instrukcji wyjścia (i odpowiednio zmiennych w instrukcji wejścia) może być pusty. Taką instrukcję wejścia-wyjścia nazywamy sygnałem. Jest to mechanizm służący jedynie do synchronizacji procesów.

Jeżeli nie prowadzi to do niejednoznaczności, można pominąć nazwę identyfikującą ciąg wyrażeń lub zmiennych, a ponadto, gdy ciągi wyrażeń i zmiennych są jednoelementowe, można również pominąć nawiasy.



Rys. 5.1. Symetryczne spotkanie w CSP

Instrukcje wejścia mogą występować w dozorach. Instrukcja wejścia postaci Q?... występująca w dozorze procesu P ma wartość false, jeśli proces Q już nie istnieje, oraz wartość true, jeśli proces Q czeka na wykonanie odpowiedniej instrukcji wyjścia P! ... — wyliczenie tego dozoru wiąże się wtedy z jednoczesnym wykonaniem instrukcji wejścia w procesie P i instrukcji wyjścia w procesie Q. Jeśli proces Q nie czeka na wykonanie swojej instrukcji wyjścia, obliczenie dozoru w procesie P zawiesza się do chwili, gdy proces Q dojdzie do tej instrukcji. W jednym dozorze może wystąpić tylko jedna instrukcja wejścia i musi być ona ostatnią częścią dozoru. To ograniczenie wynika z faktu, że nie ma możliwości cofnięcia operacji wejścia-wyjścia, gdy następujący po instrukcji wejścia warunek jest fałszywy. Jeżeli wszystkie dozory w instrukcji alternatywy lub instrukcji pętli zawierają instrukcję wejścia i w danej chwili żadna z nich nie może być wykonana, to wykonanie całej instrukcji alternatywy lub pętli zawiesza się do czasu, gdy którąś z instrukcji wejścia będzie można wykonać. Jeśli w danej chwili można wykonać więcej niż jedną instrukcję wejścia w dozorach, to do wykonania jest wybierana niedeterministycznie jedna z nich. Zakładamy przy tym, że niedeterminizm ten jest realizowany w sposób, który zapewnia własność żywotności. Oznacza to, że jeśli instrukcja alternatywy lub pętli jest wykonywana dostatecznie wiele razy i za każdym razem można wykonać daną instrukcję wejścia, to w końcu instrukcja wejścia zostanie wykonana.

5.1.5 Deklaracje

Wyróżniamy cztery standardowe typy: integer, real, boolean i char oraz typ tablicowy. Przykładowo, i: integer jest deklaracją zmiennej całkowitej i, a x:(l..N)char jest deklaracją jednowymiarowej tablicy typu znakowego.

Deklaracja może wystąpić w dowolnym miejscu treści procesu.

Obowiązuje ona od miejsca wystąpienia aż do końca tej instrukcji złożonej, w której wystąpiła (jeśli występuje na najwyższym poziomie, to obowiązuje do końca treści procesu).

5.1.6 Ograniczenia

Dwa istotne ograniczenia utrudniają programowanie w CSP. Po pierwsze, nazwy wszystkich procesów, z którymi komunikuje się dany proces, muszą być znane przed wykonaniem programu, a wiec muszą dać się wyznaczyć statycznie. Oznacza to np., że nie można napisać instrukcji wyjścia postaci P(i)!x, w której i jest wartością wyliczaną w procesie. Można ten problem ominąć korzystając ze sparametryzowanych instrukcji dozorowanych

```
[(j:1..N) i = j -> P(j)!x]
```

przy czym j jest zmienną związaną. W instrukcji alternatywy tylko ten dozór jest spełniony, w którym j = i.

Drugim utrudnieniem jest brak możliwości umieszczenia w dozorze instrukcji wyjścia. To ograniczenie ma zapobiegać nadmiernemu niedeterminizmowi programów mogącemu utrudniać ich zrozumienie i dowodzenie poprawności (por. [Bern80]). Gdyby instrukcja wyjścia mogła wystąpić w dozorze, wówczas niedeterministyczny wybór mógłby objąć nie jedną, ale jednocześnie wiele instrukcji dozorowanych w wielu różnych procesach. Na przykład w programie postaci

```
[ A::  ...   [B!x ->  ...    []  C?y  ->  ...  ]
||B::  ...   [C!z ->  ...    []  A?u  ->  ...  ]
||C::  ...   [A!v ->  ...    []  B?w  ->  ...  ] ]
```

decyzja, które z przypisań u := x, w := z czy y := v ma być wykonane, obejmuje aż trzy procesy.

Są jednak przypadki, w których możliwość wykonania instrukcji wyjścia powinna decydować o wyborze pewnego ciągu instrukcji. Ponieważ instrukcji wyjścia nie można umieścić w dozorze, należy odpowiadającą jej instrukcję wejścia w procesie-odbiorcy poprzedzić wysłaniem sygnału informującego nadawcę o gotowości odbioru. Odbiór takiego sygnału można wówczas umieścić w dozorze w procesie-nadawcy. Tę technikę stosujemy np. w przykładzie 5.2.2.

5.2 Przykłady

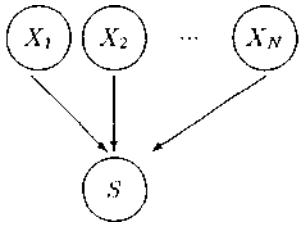
5.2.1 Wzajemne wykluczanie

Jest kilka sposobów rozwiązania problemu wzajemnego wykluczania. Najprostszy polega na zrealizowaniu w CSP semafora. Podamy tu realizację semafora ogólnego i binarnego. W przykładzie 5.2.3 pokazujemy, jak w CSP można zrealizować, monitor.

W systemach rozproszonych nie stosuje się zwykle globalnych semaforów czy monitorów do zapewnienia wzajemnego wykluczania. Rozproszony sprzęt wymaga rozproszonego oprogramowania, co oznacza rozproszenie zarówno danych, jak i sterowania. Przy takim podejściu decyzja o dostępie do sekcji krytycznej musi być podejmowana lokalnie na podstawie informacji otrzymanej od innych procesów. Przedstawimy tu dwa różne podejścia. Jedno polega na przekazywaniu uprawnień, drugie na wzajemnym dogadywaniu się procesów.

Semafor ogólny

Procesy X(1..N) korzystają z procesu semafora S wykonując instrukcje wyjścia S!V() i S!P(), które odpowiadają operacjom F i P na semaforze. W tym przypadku wystarczy wysianie samego sygnału, ponieważ semafor służy jedynie do synchronizacji i nie trzeba przesyłać żadnych danych między procesami X(i) a semaforem S. Proces S czeka bezwarunkowo na spotkanie z każdym spośród procesów X(i), i = l, ..., N, który chce wykonać S! V(), a w przypadku, gdy wartość zmiennej lokalnej s jest dodatnia, także na każdy proces X(i), i = l, ..., N, który chce wykonać S!P(). Jeśli kilka procesów X(i) będzie chciało jednocześnie wykonać instrukcje wyjścia, to w sposób niedeterministyczny zostanie wybrany jeden z nich. Ta implementacja odpowiada więc klasycznej definicji semafora podanej przez Dijkstrę. Schemat komunikacji między procesami X i procesem S przedstawia rys. 5.2.



RYS. 5.2. Semafor w CSP — schemat komunikacji

```
comment N  -  liczba procesów;
[ X(i:l..N):: ... S!V(); ...  S!P(); ...
||S:: s: integer;
   s := 0;
   *[ (i:  1..N) X(i)?V() -> s  := s + 1;
    []   (i:  1..N) s >  0;  X(i)?P()  -> s :=  s  - l
    ]
]
```

Semafor binarny

Implementacja semafora binarnego może być następująca (zgodnie z przyjętą definicją podnoszenie podniesionego semafora binarnego jest błędem):

```
SBIN::  s:  boolean;
s  := true;
*[  (1:1..N) s; X(i)?P()  -> s  := false
  []  (1:1..N) X(i)?V()  -> [s -> błąd
                           [] not s  ->  s := true]
]
```

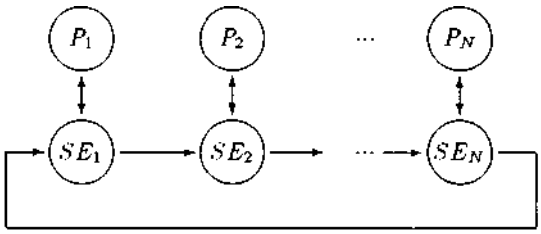
Jeśli mamy do dyspozycji proces semafor SBIN taki jak wyżej, to za jego pomocą możemy zrealizować wzajemne wykluczanie w taki oto sposób:

```
X(i:l..N)::
   *[ true -> własne_sprawy(i);
             SBINIP();
             sekcja_krytyczna(i);
             SBIN!V()
    ]
```

Przekazywanie uprawnień

Idea tego sposobu synchronizacji jest bardzo prosta. Aby uzyskać dostęp do sekcji krytycznej, proces musi otrzymać uprawnienie. Uprawnienie będzie przekazywane kolejno od procesu do procesu. Po wyjściu z sekcji krytycznej, proces powinien przekazać uprawnienie następnemu procesowi. Jednak w chwili przekazywania proces odbierający może być zajęty wykonywaniem własnych spraw, wówczas proces oddający uprawnienie byłby niepotrzebnie wstrzymywany. Dlatego z każdym procesem P związemy dodatkowy proces SEKRETARZ przyjmujący uprawnienie w imieniu P. Jemu powierzymy również przekazywanie uprawnienia dalej. Gdy proces P chce wejść do swojej sekcji krytycznej, zwraca się do swojego SEKRETARZa o zezwolenie. SEKRETARZ może pozwolić na wejście do sekcji krytycznej tylko wtedy, gdy ma uprawnienie (wskazuje to zmienna mam, początkowo uprawnienie ma SEKRETARZ(l)).

Fakt wyjścia z sekcji krytycznej proces sygnalizuje swojemu SEKRETARZowi, a ten przekazuje uprawnienie następnemu SEKRETARZowi. Schemat komunikacji między procesami P i ich SEKRETARZami przedstawia rys. 5.3.



Rys. 5.3. Przekazywanie uprawnień — schemat komunikacji

```
comment  N  -  liczba procesów;
[P(i:  1..N)::
    *[true ->  SEKRETARZ(i)!POZWÓL();
               sekcja_krytyczna(i);
               SEKRETARZ(i)!SKOŃCZYŁEM();
               własne_sprawy(i)
     ]
|| SEKRETARZ(i: 1..N):: mam:  boolean;
    mam  := i = l;
    *[mam; P(i)?POZWÓL()  ->
                P(i)?SKOŃCZYŁEM();
                SEKRETARZ(i mod N +  1)!UPRAWNIENIE();
                mam  := false
      []not mam; SEKRETARZ((i-2+N) mod  N +  1)?UPRAWNIENIE()
              -> mam  := true
     ]
]
```

Rozwiązanie to ma istotną wadę. SEKRETARZ ma uprawnienie tak długo, aż jego proces P wykona sekcję krytyczną. Oznacza to, że między dwoma kolejnymi wejściami do sekcji krytycznej każdy proces będzie musiał poczekać, aż wszystkie pozostałe procesy wejdą do swoich sekcji krytycznych.

Wymusza się w ten sposób jednakową częstość wchodzenia do sekcji krytycznej. Co gorsza, zapętlenie się jednego procesu powoduje zablokowanie pozostałych. Aby tego uniknąć, SEKRETARZ, który otrzymał uprawnienie, a którego proces P jest zajęty własnymi sprawami, będzie przekazywał uprawnienie następnemu SEKRETARZowi. Jeśli żaden proces P nie żąda dostępu do sekcji krytycznej, uprawnienie będzie ustawicznie przekazywane między SEKRETARZami. W tym rozwiązaniu SEKRETARZ musi wiedzieć, czy proces chce

wejść, do sekcji krytycznej, zatem proces P po przesłaniu sygnału POZWÓL(), czeka jeszcze na zezwolenie (sygnał MOGĘ()).

```
[P(i:  1..N)::
   *[true -> SEKRETARZ(i)!POZWÓL();
             SEKRETARZ(i)?MOGĘ();
             sekcja_krytyczna(i);
             SEKRETARZU) !SKOŃCZYŁEM() ;
             własne_sprawy(i)
   ]
|| SEKRETARZ(i:  1..N)::  chcę: boolean;
    [i = l -> SEKRETARZ(2)!UPRAWNIENIE()
    []i  <> l ->  skip] ;
   chcę  := false;
     *[P(i)?POZWÓL() -> chcę  := true
      []SEKRETARZ((i-2+N) mod N + 1)?UPRAWNIENIE()  ->
               [chcę -> P(i)!MOGĘ();
                        P(i)?SKOŃCZYLEM();
                        chcę  := false
               []not chcę  ->  skip
               ];
               SEKRETARZU mod N + 1) !UPRAWNIENE()
    ]
]
```

Opisany sposób synchronizacji procesów w systemach rozproszonych jest znany w literaturze pod nazwą token passing, czyli przekazywanie znacznika. Stosuje się go na przykład w sieciach lokalnych typu token bus, w których wszystkie komputery są połączone wspólną szyną. Podana tu implementacja jest bardzo uproszczona, gdyż nie uwzględnia możliwości zaginięcia uprawnienia ani awarii komputera, na którym wykonuje się jeden z procesów.

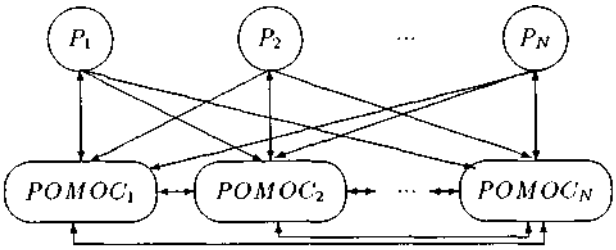
Algorytm Ricurta i Ayrawali

W roku 1981 Ricart i Agrawala zaproponowali inny rozproszony algorytm wzajemnego wykluczania [RiAgSl]. Według tego algorytmu każdy proces, który chce wejść do swojej sekcji "krytycznej, prosi o pozwolenie wszystkie pozostałe procesy. Gdy je uzyska, może bezpiecznie wykonać swoją sekcję krytyczną. Proces, który otrzymał od innego procesu prośbę o pozwolenie, postępuje różnie zależnie od tego, czy sam czeka na pozwolenia. Jeśli nie czeka, to odpowiada pozytywnie. Jeśli sam wcześniej wysłał prośby i jeszcze nie uzyskał wszystkich odpowiedzi, to o tym, kto pierwszy wykona swoją sekcję krytyczną, decyduje czas wysłania próśb. Jeśli zapytywany proces wysiał później swoją prośbę, to także odpowiada pozytywnie. Jeśli procesy wysłały prośby dokładnie w tej samej chwili, to o kolejności decydują priorytety, np. jeśli zapytywany proces ma wyższy numer, to także odpowiada pozytywnie. W każdym innym przypadku zapytywany proces wstrzymuje się z odpowiedzią aż do chwili, gdy sam skończy wykonywać swoją sekcję krytyczną. Wówczas odpowiada pozytywnie wszystkim, od których dostał zapytanie, a jeszcze im nie odpowiedział.

Podajemy teraz realizację tego algorytmu w CSP, przy czym nie uwzględniamy tu korygowania czasu (zakładamy, że funkcja czas_bieżący zwraca poprawny czas). Zasady korygowania czasu są omówione w zadaniu 5.3.5.

Z każdym procesem P jest związany proces POMOCNIK, który będzie w jego imieniu odbierał prośby o pozwolenie i odpowiednio odpowiadał na nie. Wyodrębnienie procesu POMOCNIK jest w tym przypadku konieczne, aby uniknąć blokady. Mogłaby ona wystąpić, gdyby dwa procesy zapragnęły jednocześnie wysłać do siebie prośby o pozwolenie. W tablicy

wstrzymaj każdy proces zapamiętuje numery tych procesów, od których otrzymał zapytanie i musi im odpowiedzieć po wyjściu z sekcji krytycznej. Warto zwrócić uwagę na charakterystyczny dla CSP sposób jej inicjowania. Schemat komunikacji między procesmi P i ich POMOCNIKami przedstawia rys. 5.4.



Rys. 5.4. Algorytm Ricarta i Agrawali

```
comment N - liczba procesów;
[P(i:l..N)::  t, j: integer;
  *[true -> własne_sprawy(i):
            (t, j) :=  (czas_bieżący,  1);
            POMOCNIKU) !CHCĘ(t);
            *[(k:l..N)  j  =  k  ->
                   [k  <>  i  ->  POMOCNIK(k)!CHCĘ(t)
                   []  k = i -> skip ] ;
                   j  := J + l
            ];
            POMOCNIK(i)?JUŻ();
            sekcja_krytyczna(i);
            POMOCNIKU) ! ZWALNIAMO
   ]
 ||POMOCNIK(i:l..N)::  licz,  t,  mójt: integer;
                       samchce:  boolean;
                       wstrzymaj: (1..N)  boolean;
  samchce  := false;
  *[(j:l..N) wstrzymaj(j) ->  wstrzymaj(j)  := false];
  *[(j:l..N) P(j)?CHCĘ(t) ->
       [i = j ->  (samchce, mójt, licz) := (true,  t,  0)
       [] i <>  j  ->
           [ not  samchce ->  POMOCNIK(j)!PROSZĘ()
           [] samchce;  mójt>t;  -> POMOCNIK(j)!PROSZĘ()
           [] samchce;  mójt=t;  i>j -> POMOCNIK(j)!PROSZĘ()
           [] samchce;  mójt=t;  i<j -> wstrzymaj(j)  := true
           []  samchce; mójt<t -> wstrzymaj(j)  := true
           ]
        ]
     [](j:l..N) POMOCNIK(j)?PROSZĘ() ->
         [licz = N - 2 ->  P(i)!JUŻ()
         []  licz O N - 2  -> licz := licz  +  l
         ]
     []  P(i)?ZWALNIAM -> samchce := false;
         *[(j:l..N) wstrzymaj(j) -> POMOCNIK(j)!PROSZĘ();
                                    wstrzymaj(j) := false
         ]
     ]
]
```

Warto zwrócić uwagę na to, że proces P (i) wysyła komunikat CHCĘ(t) najpierw do swojego POMOCNIKa, a dopiero potem do pozostałych. Gdybyśmy upraszczając zapis umieścili wysyłanie tego komunikatu w jednej pętli

```
*[(k:l..N)  j  = k -> POMOCNIKU) !CHCĘ(t);  j  := j + 1],
```

mogłoby to prowadzić do błędnego wykonania. Byłoby tak wówczas, gdyby jakiś proces P(j) wysłał do procesu POMOCNIK(i) komunikat CHCĘ(tl), zanim dotrze tam komunikat CHCĘ(t) od procesu P(i), przy czym tl < t.

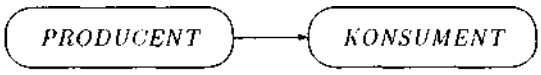
POMOCNIK(i) odpowiedziałby wówczas procesowi POMOCNIK(j) komunikatem PROSZĘO, podczas gdy zgodnie z algorytmem proces P(j) powinien w takiej sytuacji poczekać na P(i). Cały problem wynika z faktu, że decyzja o wysłaniu komunikatu CHCĘ(t) zapada w procesie P, a decyzja o pozwoleniu na skorzystanie z sekcji krytycznej w procesie POMOCNIK. Przypominamy, że podziału na dwa typy procesów dokonaliśmy po to, aby uniknąć blokady.

5.2.2 Producent i konsument

Mechanizm spotkań umożliwia rozwiązanie problemu producenta i konsumenta bez jakiegokolwiek bufora. Typ porcja oznacza tu typ przekazywanego elementu.

```
 C  PRODUCENT:: p: porcja;
   *[true  -> produkuj(p); KONSUMENT!p];
 ||KONSUMENT:: p: porcja;
   *[PRODUCENT?? -> konsumuj(p)]
]
```

Porcja jest tu przekazywana „z ręki do ręki". Schemat komunikacji między producentem a konsumentem przedstawia rys. 5.5.

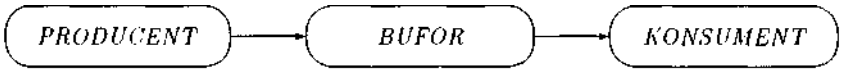


Rys. 5.5. Producent i konsument bez bufora

Ponieważ w CSP procesy nie mogą komunikować się przez wspólną pamięć, rozwiązanie z buforem wymaga wprowadzenia dodatkowego procesu BUFOR komunikującego się bezpośrednio z procesami PRODUCENT i KONSUMENT. Oto rozwiązanie z buforem jednoelementowym:

```
[ PRODUCENT::  p:  porcja;
   *[true  ->  produkuj(p);  BUFOR!p];
||BUFOR::  p:  porcja;
   * [PRODUCENT?p  -> KONSUMENT!p]
||KONSUMENT::  p:  porcja;
   *[BUFOR?p  -> konsumuj(p)]
]
```

Zauważmy, że w stosunku do poprzedniego rozwiązania procesy PRODUCENT i KONSUMENT różnią się tylko instrukcjami wejścia-wyjścia. Schemat komunikacji przedstawia rys. 5.6.



Rys. 5.6. Producent i konsument z buforem

Może się wydawać, że w rozwiązaniu z buforem cyklicznym N-elementowym treści procesów PRODUCENT i KONSUMENT powinny być takie same jak wyżej. Tak jednak nie jest, ponieważ proces BUFOR musi w jakiś sposób zdecydować, kiedy wykonać instrukcję wyjścia, a kiedy wejścia. Do zdecydowania się na instrukcję wyjścia nie wystarczy warunek, że bufor jest niepusty (z < do), proces BUFOR mógłby bowiem oczekiwać na spotkanie z procesem KONSUMENT wtedy, gdy jest on zajęty konsumowaniem, a w tym samym czasie PRODUCENT nie mógłby nic włożyć do bufora. Zmniejszyłoby to wykorzystanie bufora i niepotrzebnie wstrzymywało producenta. Dlatego trzeba wprowadzić dodatkowy sygnał JESZCZEO, który proces KONSUMENT wysyła dopiero wtedy, gdy jest już gotowy na przyjęcie nowej porcji z bufora.

W następującym rozwiązaniu użyto wskaźników do i z o takiej własności, że reszta z dzielenia modulo N wskazuje odpowiednio pierwsze wolne miejsce w buforze, do którego można wstawiać, i pierwsze zajęte miejsce w buforze, z którego można pobierać. Relacja do = z oznacza, że bufor jest pusty, a relacja do = z + N — że jest pełny. Zaniedbujemy tu możliwość przekroczenia zakresu zmiennych całkowitych do i z.

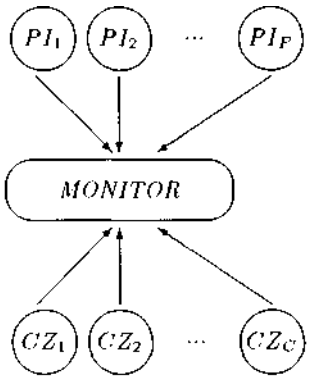
```
[ PRODUCENT:: p: porcja;
   *[true -> produkuj(p);
             BUFOR!p
   ]
||KONSUMENT:: p: porcja;
   *[true -> BUFOR!JESZCZE() ;
             BUFOR?p;
             konsumuj(p)
   ]
|| BUFOR:: bufor: (0..N-l)  porcja;
 comment N - rozmiar bufora;
   do,  z: integer;
   (do, z)   := (O,  0);
   comment O <= z  <=  do  <= z+N;
   *[do < z + N; PRODUCENT?bufor(do mód N) ->
                 do := do  + l
    []z < do; KONSUMENT?JESZCZE()  ->
                 KONSUMENT!bufor(z  mód N);
                 z := z  +  l
   ]
]
```

Inne sposoby implementacji bufora w systemie rozproszonym omawiamy w zadaniu 5.3.1.

5.2.3 Czytelnicy i pisarze

Przedstawimy tu rozwiązanie scentralizowane, w którym synchronizacja procesów CZYTELNIKÓW i PISARZy odbywa się za pośrednictwem specjalnego procesu MONITOR zarządzającego czytelnią. Na tym przykładzie omówimy ogólne zasady realizacji monitora w CSP. Rozwiązanie rozproszone tego problemu jest przedmiotem zadania 5.3.2.

Zakładamy tu, że w systemie działa C czytelników i P pisarzy. Schemat komunikacji między pisarzami i czytelnikami a procesem MONITOR przedstawia rys. 5.7.



Rys. 5.7. Czytelnicy i pisarze — komunikacja przez monitor

Rozwiązanie z priorytetem czytelników

```
comment C - liczba czytelników,
       P - liczba pisarzy;
[CZYTELNIKU:  1..C)::
 *[true -> MONITOR!POCZĄTEK_CZYTANIA();
           czytanie(i);
           MONITOR!KONIEC_CZYTANIA();
           myślenie(i)
  ]
||PISARZ(i: 1..P)::
 *[true -> myślenie(i);
           MONITOR!POCZĄTEK_PISANIA();
           pisanie(i);
           MONITOR!KONIEC_PISANIA()
  ]
||MONITOR:: c:  integer;
  comment c  -  liczba  czytających czytelników;
  c  := 0;
  *[  (i:1..C)  CZYTELNIK(i)?POCZATEK_CZYTANIA()  ->
                                                  c := c + l
    [](i:1..C)  CZYTELNIK(i)?KONIEC_CZYTANIA()  -> c := c - l
    D(i:1..P)  c  = 0;  PISARZ(i)?POCZ4TEK_PISANIA() ->
                              PISARZU) ?KONIEC_PISANIA()
   ]
]
```

Gdy jakiś czytelnik czyta, pisarze są w naturalny sposób wstrzymywani podczas wysyłania sygnału POCZĄTEK_PISANIA(), albowiem wówczas warunek c=0 w trzecim dozorze nie jest spełniony i nie dochodzi do wykonania komplementarnej instrukcji wejścia w procesie MONITOR. Jednocześnie jednak czytelnicy wysyłający sygnał POCZĄTEK_CZYTANIA() nie są wstrzymywani. Zatem czytelnicy mają tu priorytet.

Rozwiązanie poprawne

Proces, który zaczyna czekać na dostęp do czytelni, wysyła do procesu MONITOR sygnał CHCĘ_CZYTAĆ() lub CHCĘ_PISAĆ(), a potem oczekuje, aż otrzyma od niego sygnał MOGĘ_CZYTAĆ() lub MOGĘ_PISAĆ(). MONITOR musi zapamiętywać, które procesy czekają (służą do tego tablice czytelnik oraz pisarz), aby wysłać, sygnał tylko do nich. Po zakończeniu pisania, jeżeli oczekują i czytelnicy, i pisarze, wybór procesów, które otrzymają prawo dostępu do czytelni, jest niedeterministyczny, co gwarantuje, że nie ma tu możliwości zagłodzenia ani pisarzy, ani czytelników.

```
comment C  - liczba czytelników,
         P  - liczba pisarzy;
 [CZYTELNIKU: 1..C):: *[true -> MONITOR! CHCĘ_CZYTAĆ();
                                 MONITOR?MOGĘ_CZYTAĆ();
                                 czytanie(i);
                                 MONITOR!KONIEC_CZYTANIA();
                                 myślenie(i)
                        ]
 MPISARZCi: 1..P)::   *[true->  myślenie(i);
                                 MONITOR!CHCĘ.PISAĆ();
                                 MONITOR?MOGĘ_PISAĆ();
                                 pisanie(i);
                                 MONITOR!KONIEC_PISANIA()
                        ]
 || MONITOR:: c, op, p: integer;
          comment  c   -  liczba czytających czytelników,
                   op  -  liczba oczekujących pisarzy,
                   p   -  liczba piszących pisarzy;
           (c, op, p)  :=  (O,  O, 0);
          czytelnik: (1..C)  boolean;
          pisarz: (1..P)  boolean;
          comment - w  tablicach tych  pamiętamy,
                    czy  odpowiedni proces  czeka;
 *[(i:1..C) czytelnik(i)  -> czytelnik(i) := false];
 *[(i:1..P) pisarz(i)     -> pisarz(i)  := false];
 *[(i:1..C) CZYTELNIK(i)?CHCĘ_CZYTAĆ()  ->
           C op +  p  = O -> c := c +  1;
                       CZYTELNIKU) !MOGĘ_CZYTAĆ()
           []op +  p  > O -> czytelnik(i)  := true
          ]
  [](i:1..C) CZYTELNIKU)?KONIEC_CZYTANIA() ->
          c  := c  -  1;
           C(j:1..P) c = 0; pisarz(j)  ->
                  (pisarz(j), op, p)  :=  (false, op-1,  1);
                  PISARZ(j)!MOGĘ_PISAĆ()
           []c > O -> skip
          ]
  [](i:l..P) PISARZ(i)?CHCĘ_PISAĆ()  ->
           [ c + p = O -> p  := 1;
                     PISARZU) !MOGĘ_PISAĆ()
           []c + p > O -> op := op  +  1;
                     pisarzU)  :=  true
          ]
  [](i:l..P) PISARZ(i)?KONIEC_PISANIA() ->
          P  := 0;
           [(j:1..C) czytelnik(j) ->
                    *[(j:l..C) czytelnik(j) ->
                         (c, czytelnik(j))  :=  (c+1, false);
                        CZYTELNIK(j)!MOGĘ.CZYTAĆ()
                     ]
```

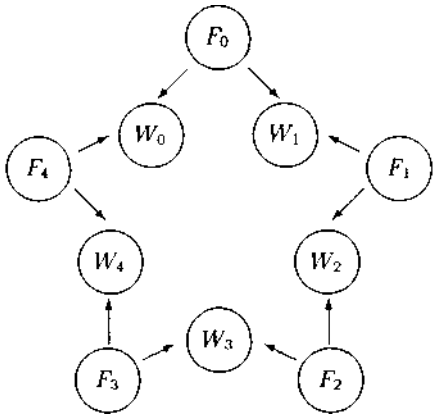
```
            [](j:1..P) pisarz(j) ->
                    (op, p, pisarz(j))  := (op-1,  1, false);
                    PISARZ(j)!MOGĘ_PISAĆ()
            ]
    ]
]
```

Warto tu zwrócić uwagę na charakterystyczny dla CSP sposób inicjowania tablic czytelnik i pisarz. Zauważmy, że proces MONITOR nigdy nie czeka podczas wykonywania instrukcji wyjścia, gdyż procesy, których dotyczą te instrukcje, są zawsze w trakcie wykonywania komplementarnych instrukcji wejścia.

Przedstawione rozwiązanie jest w istocie poglądową implementacją monitora w CSP. Oczekiwanie procesu na zezwolenie korzystania z zasobu odpowiada oczekiwaniu w kolejce związanej z warunkiem (zmienną typu condition), natomiast oczekiwanie na spotkanie z procesem MONITOR odpowiada oczekiwaniu na dostęp do monitora. Nie uwzględnia ono jednak kolejności wznawiania wstrzymanych procesów oraz kolejności „wpuszczania" procesów do monitora (w klasycznej definicji monitora zakłada się, że w obu przypadkach jest to kolejka prosta, czyli zgodna z kolejnością wstrzymywania albo wywoływania procedur monitora). Pierwszy mankament łatwo da się wyeliminować (pozostawiamy to jako zadanie czytelnikowi), drugi natomiast nie, ponieważ kolejność wpuszczania do monitora zależy wyłącznie od implementacji spotkania i niedeterministycznego wyboru.

5.2.4 Pięciu filozofów

Rozwiązanie z możliwością blokady



RYS. 5.8. Pięciu filozofów — każdy widelec oddzielnym procesem

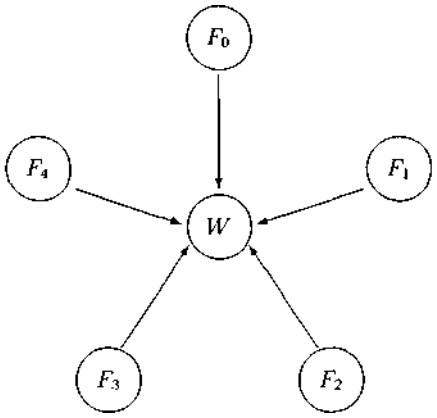
```
[FILOZOF(i: 0..4)::
   *[true -> myśli(i);
             WIDELEC(i)!WEŹ();
             WIDELEC((i+l) mod 5)!WEŹ();
             je(i);
             WIDELEC(i)!ODDAJ();
             WIDELEC((i+l) mod 5)!ODDAJ()
   ]
||WIDELEC(i: 0..4)::
   *[ FILOZOF(i)?WEŹ() -> FILOZOF(i)?ODDAJ()
    []FILOZOF((i+l)  mod 5)?WEŹ()  ->
```

```
                    FILOZOF((i+l) mod 5)?ODDAJ()
    ]
]
```

Każdy proces WIDELEC pełni rolę semafora binarnego. Podnoszenie widelca realizowane wysianiem sygnału WEŹ O odpowiada opuszczeniu semafora, a odłożenie widelca realizowane wysłaniem sygnału ODDAJ() — podniesieniu semafora. Schemat komunikacji między procesami WIDELEC a procesami FILOZOF przedstawia rys. 5.8.

Rozwiązanie z możliwością załodzenia

Proces WIDELCE zarządza wszystkimi widelcami naraz. W tablicy logicznej jedzenie pamięta się, który filozof je. Jeżeli obaj sąsiedzi filozofa żądającego widelców nie jedzą, może on otrzymać oba widelce. Schemat komunikacji przedstawia rys. 5.9.



RYS. 5.9. Pięciu filozofów — widelce jednym procesem

```
[FILOZOF(i:  0. .4) : :
   *[true -> myśli(i);
        WIDELCE! WEŹ();
        je(i);
        WIDELCE!ODDAJ();
    ]
||WIDELCE::  jedzenie:  (0..4)  boolean;
   *[(i: 0..4)  jedzenie(i)  -> jedzenie(i)  := false];
   *[(i: 0..4)  not jedzenie((i - 1) mod 5);
          not jedzenie((i  + 1) mod  5);  FILOZOF(i)?WEŹ() ->
                    jedzenie(i) := true
     [](i: 0..4) FILOZOF(i)?ODDAJ()  ->  jedzenie(i)  :=  false
    ]
]
```

5.3 Zadania

5.3.1 Producent i konsument z rozproszonym buforem

Zapisz rozwiązanie problemu producenta i konsumenta z buforem N-elementowym tak, aby każdy element bufora był reprezentowany przez odrębny proces (taki wariant ma

praktyczne uzasadnienie w sytuacji, gdy pamięć lokalna procesora wykonującego proces bufora jest na tyle mała, że mieści tylko jedną porcję). Uwzględnij dwie możliwości:

1. kolejność umieszczania wyprodukowanych elementów w buforze oraz kolejność pobierania nie mają znaczenia;
2. pobieranie elementów powinno odbywać się w takiej kolejności, w jakiej były umieszczane w buforze.

5.3.2 Powielanie plików

Jedną z typowych metod skracania czasu dostępu do informacji w systemach rozproszonych jest powielanie danych, czyli przechowywanie ich w wielu kopiach. Dostęp do lokalnej kopii jest szybszy, gdyż nie obciąża sieci komunikacyjnej. Do zadań rozproszonego systemu operacyjnego należy automatyczne powielanie plików, gdy zachodzi taka potrzeba, a zarazem ukrywanie tego faktu przed użytkownikiem, który powinien korzystać z plików tak, jak gdyby były one tylko w jednym egzemplarzu. W tym celu odpowiednie fragmenty rozproszonego systemu zarządzania plikami wykonujące się na różnych maszynach muszą synchronizować się między sobą, aby zapewnić zgodność informacji przechowywanej w poszczególnych kopiach. Wielu procesom można pozwolić na jednoczesne odczytywanie swojej lokalnej kopii, ale tylko jednemu można pozwolić na zmienianie zawartości wszystkich kopii. Mamy tu zatem typowy problem czytelników i pisarzy z tym, że w wersji rozproszonej.

1. Zsynchronizuj pracę rozproszonych czytelników i pisarzy stosując algorytm przekazywania uprawnienia z przykładii 5.2.1. Uprawnienie powinno zawierać w sobie liczbę aktualnie czytających czytelników. Czytelnik zmienia odpowiednio tę liczbę nie zatrzymując uprawnienia na czas czytania, pisarz na czas pisania zabiera uprawnienie, ale tylko wtedy, gdy ma ono wartość 0.

2. Zaproponowany wyżej algorytm dopuszcza możliwość zagłodzenia pisarzy. Podaj rozwiązanie oparte na metodzie przekazywania uprawnień nie powodujące zagłodzenia żadnej z grup. (Wskazówka: wykorzystaj pomysł zawarty w drugim rozwiązaniu w p. 3.2.3.)

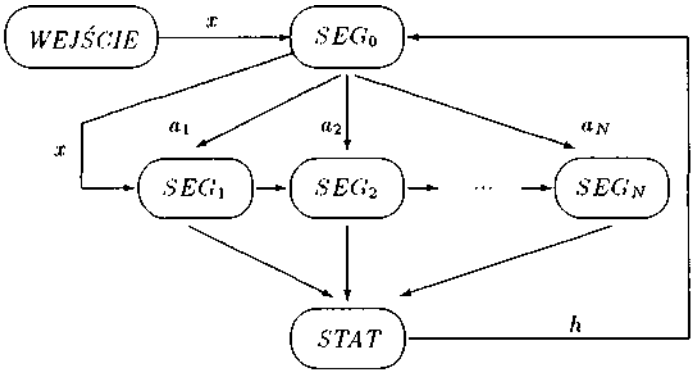
3. Zsynchronizuj pracę rozproszonych czytelników i pisarzy stosując algorytm Ricarta i Agrawali.

5.3.3 Problem podziału

Proces SPROC zarządza zbiorem liczb zbiórS o liczebności S, a proces TPROC zbiorem liczb zbiórT o liczebności T (różnych od liczb ze zbioru zbiórS).
Procesy SPROC i TPROC powinny w wyniku swojego działania spowodować umieszczenie w zbiorze zbiórS S najmniejszych liczb, a w zbiorze zbiórT T największych liczb ze zbioru zbiórS U zbiórT. Zapisz treści procesów SPROC i TPROC. Oba powinny kończyć działanie po zakończeniu podziału zbiorów.

Można założyć dostępność następujących funkcji i operacji:
max_elem(Y) - zwraca maksymalny element zbioru Y,
min_elem(Y) - zwraca minimalny element zbioru Y,
Y + {x}, Y - {x} - dodawanie (odejmowanie) elementu x do (od) zbioru Y.

5.3.4 Obliczanie histogramu



Rys. 5.10. Obliczanie histogramu — schemat komunikacji

Histogram dla pewnego ciągu danych x1,x2,...xn i wartości progowych a1, a2, ..., an jest to tablica wartości h(1..N) o tej własności, że h(i) jest mocą zbioru takich x z danego ciągu, że a(x-1) <= x < ai, dla i = 2,..., N, a h(1) jest mocą zbioru takich x, że x < a1. Współbieżne obliczanie histogramu jest realizowane przez N procesów segregujących SEG(1..N) i jeden proces statystyczny STAT. Proces użytkownika SEG(O) wysyła bezpośrednio do procesów segregujących wartości progowe z tablicy a(l. .N) tak, aby a(i) trafiło do SEG(i) (zakładamy, że zawsze a(N)= oo), a następnie wysyła do procesu SEG(l) ciąg wartości x otrzymywanych od procesu WEJŚCIE, o którym wiadomo, że kończy się po wysłaniu ostatniej wartości. Następnie proces SEG(0) wysyła sygnał końca i odbiera wyniki od procesu STAT. Proces segregujący SEG sprawdza, czy otrzymana wartość x jest mniejsza niż wartość progowa. Jeśli tak, to wysyła sygnał do procesu STAT, w przeciwnym razie przekazuje x do następnego procesu SEG. Proces STAT zlicza otrzymywane sygnały w tablicy h(l..N), którą na końcu wysyła do SEG(0).

Schemat komunikacji między omówionymi tu procesami przedstawia rys. 5.10.

Zapisz procesy SEG(0..N) i proces STAT. Procesy SEG(1..N) i STAT powinny działać cyklicznie.

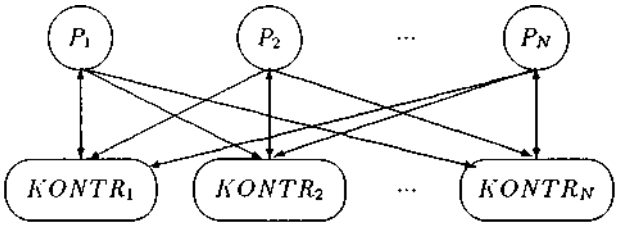
5.3.5 Korygowanie logicznych zegarów

Współbieżne procesy P(i:1..N) wykonują w nieskończonej pętli procedurę własne_sprawy(i) oraz dodatkowo co M-ty obrót pętli (począwszy od i-tego) synchronizują z innymi procesami swój logiczny lokalny zegar. Logiczny czas każdego procesu jest mierzony liczbą wykonanych obrotów pętli.

Synchronizacja jest konieczna, gdyż pętle różnych procesów wykonują się z różnymi prędkościami. Synchronizacja polega na korygowaniu lokalnego zegara na podstawie sygnowanych czasem komunikatów docierających od innych procesów. Odbieraniem komunikatów w imieniu procesu P(i) zajmuje się proces KONTROLER(i), i=l,...,N. W celu skorygowania lokalnego czasu proces P(i) komunikuje się z procesem KONTROLER(i) i

otrzymuje od niego wartość największego czasu t, jaki sygnował dotychczas odebrane komunikaty. Jeśli t jest większe od lokalnego czasu procesu, to czas ten jest ustawiany na t+1 (skoro odebrano komunikat wysłany w chwili t, to lokalny czas odbiorcy musi być późniejszy). Po skorygowaniu swojego czasu proces P (i) wysyła do jednego losowo wybranego (za pomocą funkcji losuj) procesu komunikat sygnowany własnym lokalnym czasem. Schemat komunikacji między procesami P i KONTROLER przedstawia rys. 5.11.

Zapisz procesy
P (i) i KONTROLER(i) , i=1, ..., N. Dlaczego z procesu P(i) wyodrębniono proces KONTROLER(i)?

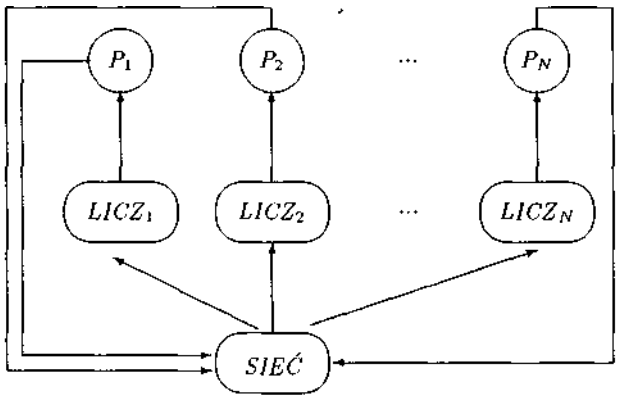


Rys. 5.11. Korygowanie logicznych zegarów

Opisany sposób synchronizowania logicznych zegarów, zaproponowany przez L. Lamporta [Lamp78], jest stosowany w algorytmach rozproszonych w celu ustalenia kolejności zdarzeń zachodzących w rozproszonym systemie, a także w algorytmie Ricarta i Agrawali omówionym w przykładzie 5.2.1. (W podanym tam rozwiązaniu nie uwzględniliśmy korygowania czasu. Czytelnikowi pozostawiamy połączenie ze sobą rozwiązań obu tych problemów.)

5.3.6 Głosowanie

Procesy P(l. .N) komunikują się w celu demokratycznego wyborom jednego z nich. Wybory odbywają się w kolejnych turach. Początkowo kandydują wszyscy. W każdej turze odpadają ci kandydaci, którzy nie otrzymają żadnego głosu, oraz ten kandydat, który uzyska najmniejszą niezerową liczbę głosów (w przypadku równej liczby głosów odpada ten proces, który ma niższy numer). Za każdym razem głosują wszystkie procesy. Głosuje się za pomocą funkcji głosuj (t), która dla N-elementowej tablicy t zwraca losowo wybrany indeks niezerowej wartości w t. Proces P (i) przekazuje swój głos procesowi SIEĆ, którego jedynym zadaniem jest rozesłanie go do wszystkich procesów biorących udział w głosowaniu. W imieniu każdego procesu P (i) głosy otrzymywane od procesu SIEĆ odbiera i zlicza proces LICZ(i), i=1, ..., N, który po otrzymaniu N głosów przekazuje tablicę wyników procesowi P (i). Schemat komunikacji między procesami P, LICZ oraz SIEĆ przedstawia rys. 5.12.



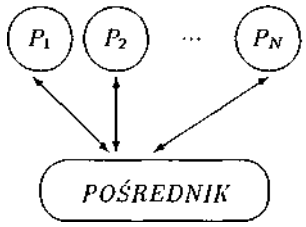
RYS. 5.12. Głosowanie — schemat komunikacji

Proces SIEĆ pełni rolę gońca roznoszącego kartki z glosami. Głosowanie odbywa się w sposób rozproszony, dlatego brak tu odpowiednika komisji skrutacyjnej. Każdy proces musi na podstawie otrzymanej informacji sam stwierdzić, jaki jest wynik głosowania w każdej turze.

Zapisz treści procesów P(i), LICZ(i), i=1, ..., N, oraz SIEĆ. Czy treść procesu SIEĆ można włączyć do procesów P(i)? Czy treść procesu LICZ(i) można włączyć do procesu P(i)?

5.3.7 Komunikacja przez pośrednika

Procesy P(l..N) współpracują ze sobą przesyłając do siebie komunikaty za pomocą procesu POŚREDNIK. U POŚREDNIKa znajduje się początkowo M komunikatów. Proces P(i), i=l, ... ,N pobiera komunikat k od POŚREDNIKa, przetwarza go wykonując procedurę przetwórz (k), wyznacza odbiorcę komunikatu realizując procedurę losuj (j), przy czym j jest numerem odbiorcy, a następnie wysyła komunikat z powrotem do POŚREDNIKa wskazując numer jego odbiorcy. Potem wykonuje procedurę wlasne_sprawy i znów oczekuje na otrzymanie komunikatu od POŚREDNIKa. Komunikaty nie muszą być odbierane od POŚREDNIKa w takiej samej kolejności, w jakiej były mu przekazane. Schemat komunikacji między procesami P i procesami POŚREDNIK przedstawia rys. 5.13. Zapisz treści procesów P(i:1..N) oraz POŚREDNIK.

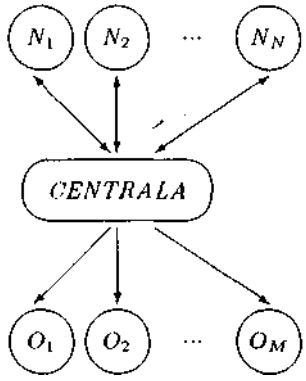


RYS. 5.13. Schemat komunikacji przez pośrednika

5.3.8 Centrala telefoniczna

W systemie działa N nadawców, M odbiorców oraz centrala, która ma K łączy (K<min(N.M)). Nadawca cyklicznie losuje numer odbiorcy wywołując procedurę losuj(m), następnie przekazuje ten numer centrali, od której otrzymuje numer łącza k. (Jeśli odbiorca jest zajęty lub brakuje wolnego łącza, to k = 0. W tym przypadku nadawca ponawia wysyłanie numeru odbiorcy do centrali.) Nadawanie jest realizowane za pomocą procedury nadaj (k). Po zakończeniu nadawania nadawca zwalnia łącze. Odbiorca czeka, aż centrala dostarczy mu numer łącza, a następnie odbiera informacje ze wskazanego łącza wywołując procedurę odbierz (k). Schemat komunikacji między procesami przedstawia rys. 5.14 (litera O oznacza odbiorcę, N — nadawcę).

Zapisz treści procesów NADAWCA(1. .N), ODBIORCA(1. .M) i CENTRALA.



RYS. 5.14. Centrala telefoniczna — schemat komunikacji

5.3.9 Obliczanie iloczynu skalarnego

Zapisz algorytm obliczenia iloczynu skalarnego dwóch N-elementowych wektorów a,b tak, aby operacje zwiększania indeksu, mnożenia dwóch współrzędnych i sumowania iloczynów realizowały się równolegle. Ostateczny wynik ma być wysłany do procesu WYJŚCIE.

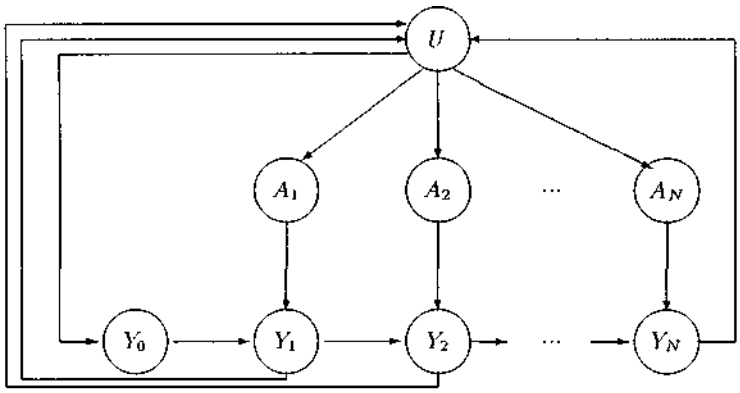
5.3.10 Obliczanie współczynników rozwinięcia dwumianu Newtona (a + b)n

Tablica procesów P(l. .K) służy do wyznaczania wierszy trójkąta Pascala. (Przypominamy, że każdy wiersz trójkąta Pascala ma na obu końcach wartość l, a każdy wewnętrzny i-ty element tego wiersza jest sumą elementów i-l i i z poprzedniego wiersza.) Użytkownik U wysyła do procesu P(l) numer wiersza, który chce otrzymać, czyli n, n < K. Następnie U odbiera współczynniki od procesów P(l), ..., P(n+l). Proces P(l) po otrzymaniu od procesu U numeru wiersza n inicjuje n-krotnie obliczanie kolejnego wiersza trójkąta Pascala. Zapisz treści procesów P(1..K) i fragment procesu U, który się z nimi komunikuje. Procesy P powinny móc wykonywać obliczenia wielokrotnie.

5.3.11 Mnożenie macierzy przez wektor

Zapisz układ procesów A(l. .N), Y(0. .N) służących do mnożenia macierzy a(l..N,l..M) przez wektor x(l. .M) oraz fragment procesu użytkownika U, który się z nimi komunikuje. Proces U wysyła najpierw wektor x element po elemencie do procesu Y(0), a następnie kolejne wiersze macierzy a również element po elemencie w ten sposób, że i-ty wiersz trafia do procesu A(i), i = l, ...,N. Poszczególne składowe wektora wyników y(l. .N) proces U odbiera od procesów Y(i), i = l,...,N. Proces Y(0) przekazuje kolejne elementy wektora x do procesu Y(l), a procesy A (i) przekazują kolejne elementy wierszy macierzy a odpowiednio do procesów Y(i), i=l, ... ,N.

Schemat komunikacji między procesami U, A i Y przedstawia rys. 5.15.



Rys. 5.15. Mnożenie macierzy przez wektor

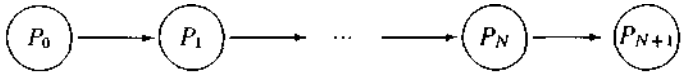
W jakiej kolejności i do kogo proces U powinien wysyłać dane, aby można było zrezygnować z procesów pośredniczących A(1. .N) i Y(0)?

5.3.12 Obliczanie wartości wielomianu

Wersja 1. Zapisz „potok" do obliczania wartości wielomianu metodą Hornera, realizowaną przez N współbieżnych procesów P(1..N). (Przypominamy, że metoda Homera polega na wykonaniu obliczeń zgodnie ze wzorem (...((aj^x + a,N-i)x + ajv_2)a; + ...)x + ao, co pozwala zaoszczędzić wielu operacji mnożenia.)

Proces P(l) dostaje od procesu użytkownika P(0) niepusty ciąg współczynników a(M), a(M-l),..,a(0), (M < N) zakończony sygnałem końca, a następnie ciąg argumentów x(l) ,. . . ,x(k) (k jest dowolne) zakończony także sygnałem końca, po którym następuje znów ciąg współczynników itd.

Proces P(N) przekazuje procesowi użytkownika P(N-H) ciąg wartości y(l), ..., y(k) itd,, przy czym y(i) = a(0) + a(l)*x(i) + . . .+a(M)*x(i)M. (Jeśli M < N - I, procesy P(M+1) , . . . ,P(N) służą jedynie do „przepychania" wyników.) Schemat komunikacji między procesami przedstawia rys. 5.16.

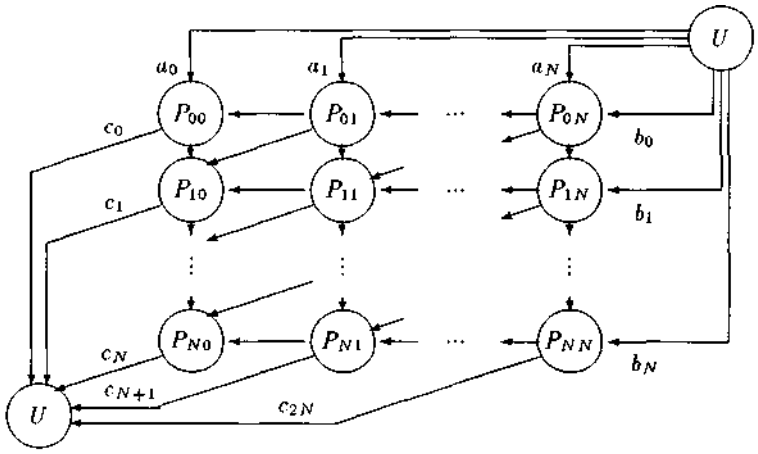


Rys. 5.16. Obliczanie wartości wielomianu

Wersja 2. Inny sposób przekazywania danych. Proces P(0) przekazuje procesowi P(l) współczynniki wielomianów w kolejności odwrotnej, a więc a(0), a(l) , ... ,a(M). Następnie zamiast pojedynczych argumentów przekazuje ciąg par liczb (x(i), 0), i = l, ...,k, zakończony sygnałem końca, potem znowu ciąg współczynników itd. Proces P(N+1) odbiera od procesu P(N) pary liczb (x(i),y(i)).

5.3.13 Mnożenie wielomianów

Tablica procesów P(0. .N,0. .N) służy do wyznaczania współczynników iloczynów wielomianów stopnia nie większego niż N. Procesy P (O, j) odbierają od procesu użytkownika U współczynniki a(j),j = 0,...,N, a procesy P(i,N) współczynniki b(i), i = O, ... ,N. Wyniki c(k) (k = O, ... ,2N) są odbierane przez proces U odpowiednio od procesów P(k,0), k = O, ... ,N, P(N,k-N), k = N+l, ... ,2N. Współczynniki a(j) są przekazywane wzdłuż kolumn w dół, b(i) zaś wzdłuż wierszy z prawa na lewo. Obliczenia wartości c(k) przebiegają wzdłuż przekątnych od prawego górnego rogu do lewego dolnego. Schemat komunikacji między procesami przedstawia rys. 5.17.



RYS. 5.17. Mnożenie wielomianów — schemat komunikacji

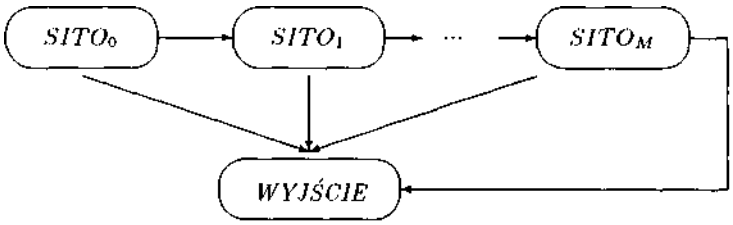
1. Zapisz algorytm procesu wewnętrznego tej tablicy, tzn. P (i, j), przy czym O < i,j < N.
2. Czym będą się różniły algorytmy procesów skrajnych?
3. Jaka jest najlepsza kolejność przekazywania współczynników do tej tablicy?

5.3.14 Sito Eratostenesa

Zapisz tablice procesów SITO (O. .M) generującą liczby pierwsze z przedziału <2, N> metodą sita Eratostenesa. Uzyskane liczby pierwsze powinny być wysłane do procesu WYJŚCIE. Przypominamy, że generowanie liczb pierwszych metodą sita Eratostenesa przebiega następująco. Najpierw wypisujemy dwójkę, a z ciągu liczb naturalnych 2,3,4,...,N wykreślamy liczby podzielne przez 2, następnie wypisujemy pierwszą niewykreśloną liczbę (będzie to oczywiście liczba 3) i wykreślamy liczby przez nią podzielne, następnie znowu

wypisujemy pierwszą niewykreśloną liczbę i wykreślamy wszystkie liczby przez nią podzielne, itd. Postępujemy tak, aż nie będziemy mogli już nic wykreślić. Wypisywane liczby i te, które pozostaną niewykreślone, to liczby pierwsze. Rysunek 5.18 przedstawia schemat komunikacji między procesami.

Jaka jest minimalna liczba sit dla danej liczby N?

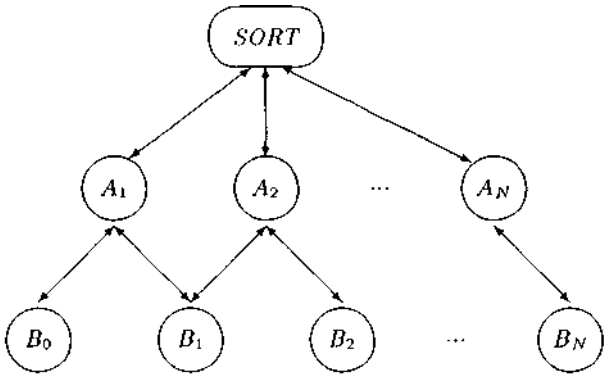


Rys. 5.18. Generowanie liczb pierwszych

5.3.15 Sortowanie oscylacyjne

Zapisz układ procesów sortujących N liczb q(1..N) złożony z dwóch typów procesów: N procesów typu A i N+1 procesów typu B.

Proces użytkownika SORT w celu posortowania N liczb, wysyła po jednej z nich do procesów A(1. .N). Proces A(i), i = 1, ... ,N, odbiera od procesu SORT liczbę, którą przekazuje do procesów B(i) i B(i-1). Następnie w pętli odbiera po jednej liczbie od procesów B(i) i B(i-1), po czym mniejszą wysyła do B(i-1), a większą do B(i). W ostatnim cyklu pętli proces A(i) dowolną z liczb otrzymanych od procesów B (powinny być równe) wysyła z powrotem do procesu SORT. (Ile razy powinna wykonać się ta pętla?) Procesy B(0) i B (N) zwracają procesom A(0) i A (N) otrzymane liczby, natomiast procesy B (i), i = 1, ... ,N-1, mniejszą z otrzymanych liczb przekazują do A(i), większą do A(i+l). Schemat komunikacji między procesami przedstawia rys. 5.19.



Rys. 5.19. Sortowanie oscylacyjne — schemat komunikacji

5.3.16 Tablica sortująca

Tablica sortująca składa się z N procesów T(1..N). Proces użytkownika T(0) przekazuje procesowi T(1) co najwyżej N liczb. W każdej chwili może też zażądać od procesu T(1) zwrotu liczby i wówczas powinien otrzymać najmniejszą ze znajdujących się w tablicy. Proces T(i), i = 1, ... ,N, pobiera od poprzednika T(i-1) liczbę. Wszystkie następne otrzymywane od T (i-1) porównuje z tą, którą już ma. Jeśli nowa liczba jest mniejsza od posiadanej, to zatrzymuje ją, a dotychczas posiadaną przekazuje do procesu T(i+1). W przeciwnym razie otrzymaną liczbę przekazuje od razu do procesu T(i+1). Gdy proces T(i-1) zażąda od T(i) liczby, przekazuje on mu tę aktualnie posiadaną i ewentualnie żąda liczby od T(i+1). Schemat komunikacji między procesami przedstawia rys. 5.20.

Rys. 5.20. Tablica sortująca — schemat komunikacji

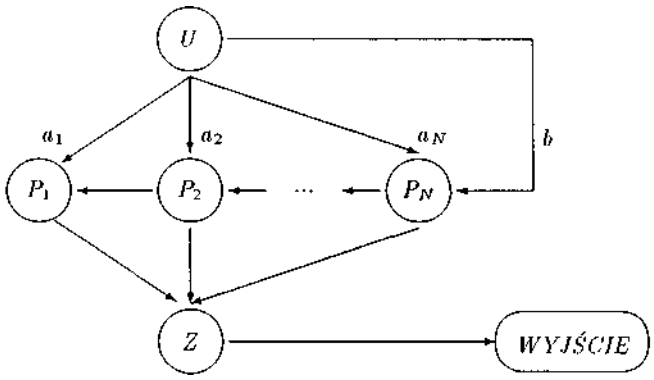
Zapisz treść tablicy procesów T(1. .N) oraz fragment T(0), w którym się z niej korzysta. (Zadanie to pochodzi od Brinch Hansena.)

5.3.17 Porównywanie ze wzorcem

Układ procesów do porównywania ze wzorcem składa się z N procesów porównujących P(l),...,P(N)i procesu zliczającego Z. Proces użytkownika U wysyła najpierw elementy wzorca a(l. .N) do procesów porównujących tak, aby wartość a(i) trafiła do procesu P(i), a następnie wysyła po kolei elementy tablicy b(1. .M), M » N, do procesu P (N). Elementy te „przepływają" kolejno przez wszystkie procesy aż do P(l). Proces P(i) porównuje każdy otrzymany element z elementem wzorca. Jeśli jest zgodność, to wysyła do Z liczbę n-i+1 (przy czym n jest numerem kolejnego elementu) gdyż wykrył częściowe dopasowanie wzorca od pozycji n-i+1. Proces Z zlicza wszystko, co dostaje od procesów P, a te liczby, które otrzymał dokładnie N razy, wysyła do procesu WYJŚCIE (wzorzec pasuje na wszystkich N pozycjach).

Schemat komunikacji między procesami przedstawia rys. 5.21.

Zapisz treści procesów P, Z i fragment U, który je uruchamia.



RYS. 5.21. Porównywanie ze wzorcem — schemat komunikacji

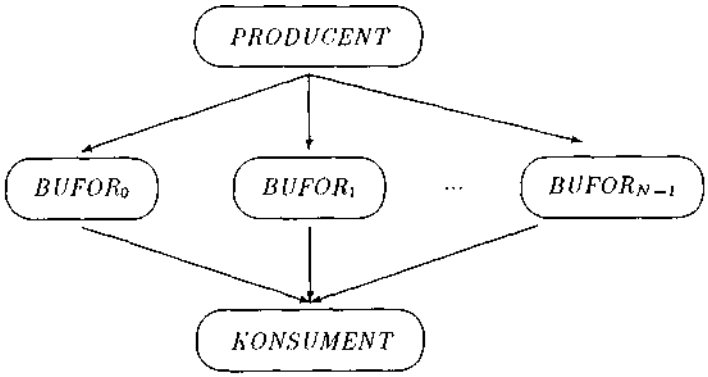
5.4 Rozwiązania

5.4.1 Producent i konsument z rozproszonym buforem

Rozwiązanie I

Oto rozwiązanie, w którym kolejność nie jest istotna. Każdy element bufora BUFOR(i), i = 0,...,N-1, informuje proces PRODUCENT o tym, że jest wolny, próbując wysiać do niego sygnał JESZCZE 0.

Proces PRODUCENT wysyła wyprodukowaną porcję do tego elementu bufora, od którego udało mu się odebrać sygnał JESZCZE Q. Schemat komunikacji między procesami przedstawia rys. 5.22.



Rys. 5.22. Producent i konsument — rozwiązanie 1 i 3

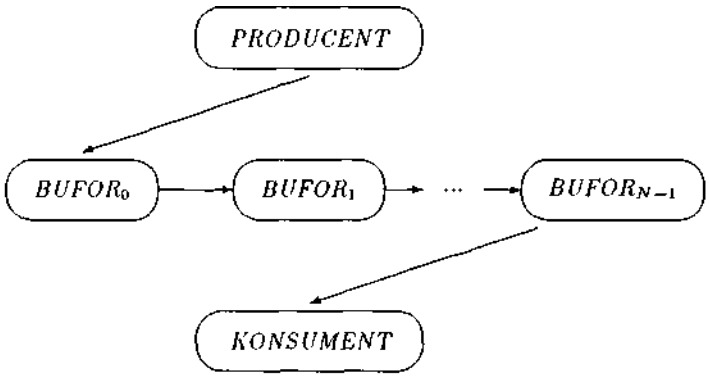
```
comment N - rozmiar bufora;
[PRODUCENT:: p:  porcja;
  *[true -> produkuj(p);
             [(i:0..N-l) BUFORU)TJESZCZEO  -> BUFOR(i)!p]
  ]
```

```
||BUFOR(i:0..N-l):: p: porcja;
  *[true -> PRODUCENT! JESZCZE() ;
            [PRODUCENT?? -> KONSUMENT!p]
  ]
||KONSUMENT:: p:  porcja;
  *[(i:0..N-l)  BUFOR(i)?p -> konsumuj(p)
]
```

Rozwiązanie 2

Oto rozwiązanie z uwzględnieniem kolejności. Porcje są „przepychane" przez kolejne elementy bufora. Proces PRODUCENT wstawia zawsze do pierwszego elementu Inifora. Proces KONSUMENT pobiera zawsze od ostatniego elementu bufora. Schemat komunikacji między procesami przedstawia rys. 5.23.

```
comment N -  rozmiar bufora;
[PRODUCENT:: p:  porcja;
   *[true  ->  produkuj(p); BUFOR(0)!p]
||BUFOR(0)::  p: porcja;
   * [PRODUCENT?? -> BUFOR(l)!p]
||BUFOR(i:l..N-2):: p: porcja;
   *[BUFOR(i-l)?p -> BUFOR(i-H)!p]
 || BUFOR(N-1)::  p:  porcja;
   *[BUFOR(N-2)?p -> KONSUMENT!?]
 || KONSUMENT::  p: porcja;
   *[BUFOR(N-l)?p -> konsumuj(p)]
 ]
```



Rys. 5.23. Producent i konsument — rozwiązanie 2

Procesy BUFOR(0..N-1) można zapisać także krócej następująco:

```
BUFOR(i:O..N-l):: p: porcja;
   *[true  -> [i = 0 -> PRODUCENT?p
              []i <> 0 -> BUFOR(i-l)?p];
              [i = N-l -> KONSUMENT!?
              []i <> N-l -> BUFOR(i+l)!p]
    ]
```

Rozwiązanie 3

W poprzednim rozwiązaniu każda porcja jest przekazywana między kolejnymi procesami bufora. Dzięki temu procesy producenta i konsumenta są bardzo proste, ale przejście porcji przez bufor może trwać długo.

Następujące rozwiązanie nie ma tej wady. Użyto tu przełącznika nast, który wskazuje procesowi PRODUCENT, do którego elementu bufora ma wstawiać, a procesowi KONSUMENT, z którego elementu bufora ma pobierać.

Schemat komunikacji między procesami przedstawia rys. 5.22.

```
comment N  - wielkość bufora;
[PRODUCENT:: p: porcja; nast:  integer;
   nast := 0;
  *[true -> produkuj(p);
        [(i: 0..N-1) nast  = i -> BUFOR(i)!p;
                                  nast  :=  (nast+1) mod  N]
   ]
||BUFOR(i:  0..N-1):: p:  porcja;
  *[PRODUCENT?? -> KONSUMENT!?]
|l KONSUMENT:: p: porcja; nast: integer;
  nast  := 0;
  *[(i:  0..N-1) nast  =  i; BUFOR(i)?p  ->
                            nast  :=  (nast+1) mod N;
                            konsumuj(p)
   ]
]
```

5.4.2 Powielanie plików

Rozwiązanie I

Przyjmiemy założenie, że każdy z procesów P(l..N) może być albo czytelnikiem, albo pisarzem. Czytelnik będzie wysyłał do swojego SEKRETARZa komunikat POZWÓL (f alse), a pisarz komunikat POZWÓL (true). Pozostałe komunikaty będą miały takie samo znaczenie jak w przykładzie 5.2.1.

Zmienna logiczna pisarz wskazuje, czy obsługa dotyczy czytelnika, czy pisarza, a zmienna logiczna mam mówi, czy czytelnik zarezerwował sobie miejsce w czytelni (dodając jedynkę do krążącego komunikatu). Zmienna chcę ma wartość true między odebraniem komunikatu POZWÓL a odebraniem komunikatu SKOŃCZYŁEM. Dzięki temu można odróżnić stan, w którym czytelnik już skończył czytać, ale ma jeszcze prawo do czytania. Jeśli w tym stanie będzie chciał ponownie czytać, to może to zrobić natychmiast. Jeśli w tym stanie otrzyma UPRAWNIENIE, to oddaje prawo do czytania. Zauważmy, że gdyby nie odróżniało się tego stanu, to kolejne czytania przez ten sam proces odbywałyby się co dwa obiegi żetonu.

```
comment N - liczba procesów;
SEKRETARZU:  1..N)::  chcę, mam, pisarz:  boolean;
                      ile:  integer;
    [i = l ->  SEKRETARZ(2)!UPRAWNIENIE(0)
    [] i <> l  -> skip ] ;
    chcę := false;
    mam  := false;
    *[P(i)?POZWÓL(pisarz)  -> chcę  :=  true;
                              []not pisarz; mam  ->  P(i)!MOGĘ()
                              []not pisarz;  not mam -> skip
```

```
                        []pisarz -> skip ]
    []SEKRETARZ((i-2+N) mod N + l)?UPRAWNIENIE(ile) ->
           [chcę; pisarz;  ile = 0 -> P(i)!MOGĘ();
                                      P(i)?SKOŃCZYŁEM();
                                      chcę  :=  false
           Gchcę; pisarz;  ile > 0 -> skip
           []chcę; not pisarz; mam -> skip
           Gchcę; not pisarz; not mam ->
                           (mam, ile)  :=  (true, ile +  1);
                           P(i)!MOGĘ()
           Gnot chcę; mam ->
                          (mam, ile) :=  (false, ile -  1);
           Gnot chcę; not mam -> skip
           ];
           SEKRETARZU mod N + 1) lUPRAWNIENE(ile) ;
    []P(i)?SKOŃCZYLEM() -> chcę := false
    ]
```

Rozwiązanie 2

Ponieważ wszyscy czytelnicy mogą odczytywać swoje kopie równocześnie, więc między procesami powinno dodatkowo krążyć tyle komunikatów, ilu jest czytelników. Komunikaty te nazwiemy MIEJSCEO, gdyż symbolizują one miejsce w czytelni. Zakładamy, że liczba czytelników (miejsc) wynosi C, C < N-l. Na początku proces SEKRETARZ(l) wyśle w pierścień C komunikatów MIEJSCEO oraz jeden komunikat UPRAWNIENIEO, który będzie pełnił taką samą rolę jak semafor binarny W w przykładzie 3.2.3, tzn. będzie służył do wzajemnego wykluczania pisarzy. SEKRETARZ czytelnika może zezwolić na czytanie, jeśli otrzyma komunikat MIEJSCEO. SEKRETARZ pisarza może pozwolić na pisanie, jeśli otrzyma komunikat UPRAWNIENIEO, a następnie zbierze wszystkie komunikaty MIEJSCEO. Po zakończeniu czytania proces SEKRETARZ generuje nowy komunikat MIEJSCEO, po zakończeniu pisania trzeba wygenerować na nowo wszystkie komunikaty. W zmiennej ile zlicza się, ile komunikatów MIEJSCEO zebrał już pisarz. Zmienna mam_prawo wskazuje, czy pisarz już zdobył uprawnienie.

```
comment N - liczba procesów,
        C - liczba czytelników, C < N -  1;
SEKRETARZU: 1..N):: chcę,  pisarz: boolean;
                    ile:  integer; mam_prawo: boolean;
     [i = l  -> SEKRETARZ(2)!UPRAWNIENIEO;
                ile  := C;
                *[ile >  O ->  SEKRETARZ(2) !MIEJSCEO;
                               ile := ile -  1];
   []i <> l -> skip
   ];
   chcę := false;
   *[P(i)?POZWÓL(pisarz)  ->
           (chcę, mam_prawo) := (true, false)
    [] SEKRETARZ (U-2+N) mod N + 1)?MIEJSCE()  ->
           [chcę; not  pisarz -> P(i)!MOGĘ();
                                 chcę  := false
           []chcę; pisarz;  mam_prawo  -> ile  := ile + 1;
                           [ile = C -> P(i)!MOGĘ();
                                       chcę  := false
                           [] ile < C  - >  skip ]
           []  chcę;  pisarz; not mam_prawo ->
                           SEKRETARZU mod N + 1)!MIEJSCE()
           []not  chcę  ->  SEKRETARZ(i  mod  N +  1) !MIEJSCE()
           ]
```

```
 [] SEKRETARZ (U-2+N) mod N + 1)?UPRAWNIENIE()  ->
        [chcę; not  pisarz ->
                    SEKRETARZU mod N + 1) !UPRAWNIENIE()
        []chcę; pisarz -> mam_prawo  := true;
        []not  chcę  ->
                    SEKRETARZU mod N + 1) !UPRAWNIENIE()
         ]
 []P(i)?SKOŃCZYŁEM()  ->
        [not pisarz -> SEKRETARZU mod N  + 1)!MIEJSCE()
        [] pisarz     ->
                   SEKRETARZU mod  N +  1)!UPRAWNIENIE();
                   *[ile > O ->
                       SEKRETARZU  mod  N  +  1)!MIEJSCE();
                       ile := ile - l
                    ]
         ]
]
```

Zauważmy, że założenie C < N - l jest istotne. Aby system działał, w pierścieniu może krążyć co najwyżej o jeden komunikat mniej niż jest procesów. Ponieważ proces SEKRETARZU) wysyła w pierścień C + 1 komunikatów, więc musi w nim być jeszcze co najmniej dwóch pisarzy. W przeciwnym razie powstałaby blokada. Jeśli system nie spełnia tego warunku, można dołożyć jednego lub dwóch SEKRETARZy (bez zwierzchniego procesu P) tylko po to, aby przechowywali krążące komunikaty.

Prezentowane rozwiązanie nie dopuszcza do zagłodzenia żadnej z grup. Wynika to z faktu, że wszystkie komunikaty krążą w pierścieniu. Zanim komunikat odwiedzi dwukrotnie ten sam proces, musi przedtem odwiedzić wszystkie pozostałe procesy, a więc każdy proces będzie miał swoją szansę zdobycia miejsca lub uprawnienia.

Rozwiązanie 3

Rozwiązanie problemu rozproszonej synchronizacji czytelników i pisarzy możemy otrzymać nieznacznie tylko modyfikując algorytm Ricarta i Agrawali.

Zauważmy, że czytelnik ma prawo odczytać swoją kopię, jeśli pozwolą mu na to wszyscy pisarze. Pisarz natomiast może zmienić wszystkie kopie, gdy pozwolą mu na to zarówno czytelnicy, jak i pozostali pisarze, a więc wszyscy współzawodniczący o dostęp. Jeżeli zatem tak ponumerujemy procesy, aby pisarze mieli numery od l do K a czytelnicy od K + l do N, to algorytm pisarzy i ich pomocników pozostaje taki, jak w algorytmie Ricarta i Agrawali, natomiast w procesach czytelników i ich pomocników trzeba odpowiednio zastąpić N przez K.

Rozwiązanie to nie preferuje żadnej z grup. Procesy są obsługiwane w takiej kolejności w jakiej zgłosiły swoje żądania z tym, że zgłaszający się po sobie czytelnicy mogą odczytywać swoje kopie jednocześnie.

5.4.3 Problem podziału

Zakończenie podziału można rozpoznać po tym, że jeden proces zwraca drugiemu liczbę wcześniej od niego otrzymaną. W następującym rozwiązaniu inicjatywę ma proces SPROC, który jako pierwszy wybiera liczbę maksymalną i przekazuje ją procesowi TPROC oraz decyduje, kiedy zakończyć działanie. TPROC odbiera jedynie liczby od SPROC, umieszcza je w swoim zbiorze zbiórT i zwraca element minimalny. Zauważmy, że proces

SPROC może usiiwać element ze zbioru zbiórS dopiero po otrzymaniu odpowiedzi od procesu TPROC.

```
comment S - liczba elementów  zbioru  zbiórS,
         T - liczba elementów  zbioru  zbiórT;
[SPROC:: zbiórS: (1..S)  integer;
          m, n:  integer;
          dalej:  boolean;
   dalej  := true;
   *[dalej -> m  := max_elem(zbiórS);
               TPROC!m;
               TPROC?n;
                [m = n ->  dalej   := false;
                [] m <> n  -> zbiórS  := zbiórS - {m} +  {n}
               ]
     ]
IITPROC::  zbiórT:  (1..t) integer;  m:  integer;
   *[SPROC?m -> zbiórT := zbiórT  + {m};
                m  := min_elem(zbiórT);
                zbiórT := zbiórT  - {m};
                SPROCIm
    ]
]
```

5.4.4 Obliczanie histogramu

W tym rozwiązaniu warto zwrócić uwagę na sposób przesyłania wartości progowych z procesu SEG(O) do procesów SEG(k), k = 1,... ,N, wynikający z faktu, że nie można użyć zmiennej jako indeksu procesu. Wynikowa tablica h jest w procesie SEG(O) zadeklarowana bezpośrednio przed jej użyciem. Po zakończeniu się procesu WEJŚCIE dozór drugiej pętli w procesie SEG(O) staje się fałszywy i proces ten wysyła sygnał KONIEC(). Sygnał ten jest przekazywany kolejno przez wszystkie procesy SEG(k:l. ,N), a proces SEG(N) przekazuje go procesowi STAT. W chwili odebrania tego sygnału proces STAT wie na pewno, że odebrał już wszystkie wysiane do niego sygnały LICZ.

W procesie STAT zastosowano charakterystyczny sposób zerowania tablicy h. Pętla wykona się dokładnie tyle razy, ile elementów niezerowych było inicjalnie w tablicy. Jeśli wszystkie deklarowane w procesie zmienne są automatycznie ustawione na wartości zerowe, to pętla nie wykona się ani razu.

Oczywiście i tak wymaga to sprawdzenia wszystkich N dozorów pętli, ale CSP jest językiem zakładającym najwyższy możliwy stopień współbieżności.

Zmienna dalej w procesach SEG i STAT służy do zatrzymania pętli w razie otrzymania sygnału końca KONIEC().

```
comment N - liczba wartości  progowych;
[SEG(O)::  i: integer; x: real;  a:  (1..N) real;

     i := 1; *[(k:l..N) k =  i  ->  SEG(k)!a(k);  i := i +  1];
     *[WEJŚCIE?x  -> SEG(l)!x];
     SEG(1)!KONIEC();
     h: (l..N)  integer;
     i := 1; *[i  <= N -> STAT?h(i); i  := i  +  1];

||SEG(k:l..N)::  a,x:real; dalej:  boolean;
   *[SEG(0)?a  ->
```

```
        dalej  := true;
        *[dalej; SEG(k-l)?x ->  [x  < a -> STAT!LICZ()
                                 []  x >=  a ->  SEG(k+l)!x
                                 ]
          [] dalej;  SEG(k-l)?KONIEC() ->
                                [k = N -> STAT!KONIEC()
                                [] k < N -> SEG(k+l)!KONIECQ
                                ]
                                dalej := false;
        ]
     ]
||STAT::  i:integer;  h:(1..N)  integer; dalej: boolean;
   *[true ->
        *[(k:l..N)  h(k)  <>  0 -> h(k) := 0];
        dalej  := true;
        *[(k:l..N)  dalej; SEG(k)?LICZ() -> h(k)  :=  h(k)  + l
          [] dalej;  SEG(N)?KONIEC() -> dalej  := false];
          i :=  1;
          *[i <= N  ->  SEG(0)!h(i); i := i + 1]
     ]
]
```

5.4.5 Korygowanie logicznych zegarów

W prezentowanym rozwiązaniu proces P chcąc otrzymać od KONTROLERa największą wartość zaobserwowanego czasu, wysyła do niego najpierw sygnał DAJ(). Wartość lokalnego logicznego zegara jest pamiętana w zmiennej c. Numer obrotu pętli, w którym ma nastąpić synchronizacja jest pamiętany w zmiennej z.

```
[P(1:1..N)::  j, z, c,  t:  integer;
      (c,  z)  :=  (O, i);
      *[true -> własne_sprawy(i);
          c  := c  + 1;
          [ c O  z ->  skip
          []  c =  z ->  KONTROLER(i)!DAJ();
                     KONTROLER(i)?t;
                     [t >  c -> c := t + l
                     []  t  <= c -> skip ]
                     z  :=  c + M;
                     losuj(j);
                     [(k:l..N) j  = k -> KONTROLER(k)!c]
          ]
      ]
||KONTROLER(i:l..N)::  c,  t: integer;
      c  := 0;
      *[  (k:l..N)  P(k)?t -> [t > c -> c := t
                              []  t <= c -> skip]
     [] P(i)?DAJ()  -> P(i)!c
     ]
]
```

Są dwa powody wyodrębnienia procesu KONTROLER (przyjmującego komunikaty) z procesu P (wysyłającego komunikaty). Po pierwsze, chodzi o uniknięcie blokady, która mogłaby powstać, gdyby pewna grupa procesów P chciała jednocześnie wysiać komunikaty ze swoimi czasami do innych procesów z tej grupy. Każdy proces chciałby wysiać, ale nikt nie

chciałby odebrać. Po drugie, w trakcie wykonywania własnych spraw proces nie mógłby przyjmować komunikatów i nadawca byłby niepotrzebnie wstrzymywany.

5.4.6 Głosowanie

Ponieważ na początku wszystkie procesy mają równe szansę w głosowaniu, w lokalnej tablicy t procesu P umieszczamy najpierw same jedynki. Po oddaniu głosu i otrzymaniu wyników głosowania proces P znajduje w tej tablicy niezerowe minimum. Głosowanie kończy się, gdy to minimum ma wartość N, co oznacza, że wszystkie glosy padły na jeden proces; zmienna min wskazuje wówczas na wybrany proces. Proces LICZ cyklicznie pobiera od procesu SIEĆ po N liczb traktując je jako głosy jednej tury, zlicza je w tablicy w i przekazuje procesowi P.

```
[P(i:l..N)::   t:(l..N) integer;
               jeszcze: boolean;
               min, j:  integer;
     *[(k:l..N) t(k) <> l ->  t(k):=  1] ;
     jeszcze := true;
     * [jeszcze -> SIEĆ!głosuj(t);
          LICZ(i)?t;
          min := 1;
          *[t(min) = O -> min := min +  1];
          j  := min + 1;
          *[j<=N -> [t(j) >=  t(min)  ->  skip
                     Gt(j)  =  O -> skip
                     []t(j)  <> 0; t (j)  < t (min) -> min  := j
                     ];
                     j - J  +  l
          ];
          jeszcze := t(min) <> N;
          t(min)  := 0;
     ]
|| LICZ(i:l..N):: k,  j:integer; w:(l..N) integer;
      *[true ->
           *[(k:l..N)  w(k) <> O  ->  w(k)  := 0];
           j :=  1;
           *[j<=N; SIEĆ?k ->  (j, w(k)) := (j+1,  w(k)+l)];
           P(i)!w
      ]
|| SIEĆ::  j, g:integer;
      *[(i:l..N) P(i)?g -> j  :=  1;
           *[(k:l..N)  j = k -> LICZ(k)!g; j  :=  j + 1]
      ]
]
```

Jak widać, proces SIEĆ jedynie rozsyła liczbę otrzymaną od procesu P do wszystkich procesów LICZ. Jest więc tylko pośrednikiem w przekazywaniu. P mógłby sam rozsyłać swoje glosy do wszystkich procesów LICZ. LICZ musiałby jednak wówczas sprawdzać, od którego procesu P otrzymał już glos w danej turze. Jest to konieczne, gdyż pewne procesy P mogą chcieć oddać glos w następnej turze (bo ich procesy LICZ już otrzymały wszystkie glosy), zanim glosy z poprzedniej tury trafią do wszystkich procesów LICZ. W rezultacie proces LICZ może otrzymywać przeplatane glosy z dwóch różnych tur.

Włączenie procesu LICZ (przyjmującego komunikaty) do procesu P (wysyłającego komunikaty) nie jest możliwe, gdyż mogłaby powstać blokada wtedy, gdyby pewna grupa procesów P chciała jednocześnie wysiać komunikaty ze swoimi głosami. SIEĆ odebrałaby

pierwszy komunikat, ale nie mogłaby przekazać go dalej, bo wszystkie inne procesy z tej grupy nie czekałyby na odbiór z SIECi lecz na wysianie do niej.

5.4.7 Komunikacja przez pośrednika

Proces POŚREDNIK przechowuje informację o komunikatach w trzech tablicach: ile: (1..N) integer; kom: (1..M) komunikat; adr: (1..M) integer. Element tablicy ile(j), j = l, . . . ,N, zawiera liczbę komunikatów znajdujących się u pośrednika, skierowanych do procesu P(j); natomiast kom(i), i=l.....M, zawiera komunikat skierowany do procesu o numerze adr (i), takim że adr(i)=j. Jeżeli POŚREDNIK ma komunikaty dla procesu P, to przyjnnije od niego żądania przesłania komunikatu. Następnie wyszukuje komunikat i przekazuje go adresatowi, modyfikując informację o komunikatach. Ponieważ w systemie jest stalą liczba komunikatów, więc jeśli jakiś proces P przekazuje POŚREDNIKowi komunikat, to na pewno jest nań wolne miejsce w tablicach. Po otrzymaniu komunikatu POŚREDNIK znajduje wolne miejsce w tablicach kom i adr, a następnie zapamiętuje komunikat wraz z niezbędnymi danymi.

```
comment N - liczba procesów,
        M - liczba komunikatów;
[P(i: 1..N)::  k:  komunikat; j: integer;
    comment j  jest numerem odbiorcy;
    *[true ->  POŚREDNIK! CHCĘ ();
               POŚREDNIK?k;
               przetwórz(k);
               losuj(j);
               POŚREDNIK!(k,j);
               własne_sprawy(i)
    ]
||POŚREDNIK::
    ile:  (1..N)  integer;
    kom:  (1..M)  komunikat; k: komunikat;
    adr:  (1..M)  integer;    i, n:  integer;
    i :=  1; *[i  <= N -> ile(i)  := 0;  i := i + 1];
    i :=  1; *[i  <= M -> losuj(adr(i));
                          ile(adr(i))  := ile(adr(i)) + 1;
                          kom(i)  := generuj.komunikat;
                          i  := i -t- l
              ];
    *[(j: 1..N)  P(j)?(k,n) -> i  :=  1;
                *[adr(i)<>0  -> i  := i  + 1];
                (adr(i), kom(i), ile(n)) := (n, k, ile(n)-H);
     [](j: 1..N)  ile(j) > 0; P(j)?CHCĘ()  ->
                          i  := 1;
                          *[adr(i)<>j  -> i := i +  1];
                          P(j)!kom(i);
                          (adr(i), ile(j)) := (O,  ile(j)-l)
    ]
]
```

W przedstawionym rozwiązaniu wyszukiwanie wolnego miejsca oraz adresu w tablicy adr odbywa się zawsze od początku. W rezultacie komunikaty każdego adresata są przez pośrednika przechowywane zgodnie z kolejką odwrotną (stos). Zmodyfikujemy proces POŚREDNIK tak, aby komunikaty były traktowane sprawiedliwie. W poniższym rozwiązaniu po przyjęciu komunikatu następuje niedeterministyczny wybór wolnego elementu tablicy adr,

natomiast po otrzymaniu od procesu P sygnału CHCĘO — niedeterministyczny wybór komunikatu skierowanego do tego procesu.

```
POŚREDNIK::
     ile:  (1..N) integer;
     kom:  (1..M) komunikat;  k:  komunikat;
     adr:  (1..M) integer;   i,  n:  integer;
     i  :=  1;  *[i <= N ->  ile(i) := 0; i  :=  i + 1] ;
     i  :=  1;  *[i <= M ->  losuj(adr(i));
                          ile(adr(i)) :=  ile(adr(i)) +  1;
                          kom(i) := generuj.komunikat;
                          i  :=  i + l
                ];
     *[(j:  1..N) P(j)?(k,n)  ->
             [(i: 1..M)  adr(i)  = O ->
                    (adr(i),  kom(i), ile(n))  := (n, k, ile(n)-H)
             ]
     [](j:  1..N) ile(j)  > 0;  P(j)?CHCĘ() ->
             [(i: 1..M)  adr(i)  = j -> P(j)!kom(i);
                    (adr(i),  ile(j))  := (O, ile(j)-l)
             ]
     ]
```

5.4.8 Centrala telefoniczna

Proces CENTRALA musi pamiętać stan swoich łączy. Zmienna wolne wskazuje, ile łączy jest w danej chwili wolnych. W tablicy kanał jest pamiętane, który z procesów ODBIORCA korzysta aktualnie z danego łącza. W tablicy odbiór pamięta się, czy dany ODBIORCA jest aktualnie zajęty inną rozmową. Centrala zwraca procesowi NADAWCA wartość O, jeśli nie ma wolnego łącza (wolne = 0) lub jeśli ODBIORCA, z którym chce się połączyć nadawca, jest zajęty (odbiór(m) = true). Jeśli połączenie jest możliwe, proces CENTRALA zmniejsza wartość zmiennej wolne i wyszukuje numer wolnego łącza przeglądając tablicę kanał. Następnie informuje odpowiedni proces ODBIORCA wysyłając do niego numer łącza, przez które ma odebrać rozmowę (telefon dzwoni u odbiorcy). Ponieważ w CSP nie można użyć zmiennej jako indeksu procesu, zastosowano tu instrukcję alternatywy z tablicą dozorów. Proces NADAWCA informuje proces CENTRALA o zakończeniu nadawania wysyłając komunikat ZWALNIAM(k), przy czym k jest numerem przyznanego łącza (oznacza to odłożenie słuchawki). W tym rozwiązaniu tylko proces NADAWCA może zwolnić linię. Odpowiada to rzeczywistemu zachowaniu się central telefonicznych.

```
  comment N - liczba nadawców,
          M - liczba odbiorców,
          K - liczba łączy;
[NADAWCA(i:l..N)::  m,  k:  integer;
  *[true -> losuj(m);
            k  :=  0;
            *[k = O -> CENTRALAIm; CENTRALA?k];
            nadaj(k);
            CENTRALA!ZWALNIAM(k);
  ]
||ODBIORCA(i:l..M)::  k: integer;
  *[CENTRALA?k -> odbierz(k)]
MCENTRALA:: m, i,  wolne: integer;
             kanał:  (1..K) integer;
             odbiór:  (1..M) boolean;
    *[(!:!..K) kanał(i) <> O -> kanał(i)  := 0];
```

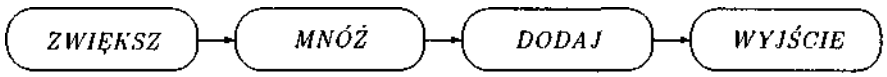
```
      *[(i:l..M) odbiór(i)   -> odbiór(i)  :=  false];
      wolne  := K;
      *[(j:l..N) NADAWCA(j)?m ->
          [wolne  = O -> NADAWCA(j)!0
          []  odbiór(m) -> NADAWCA(j)!0
          []  not  odbiór(m);  wolne O O  ->
                    (i,  wolne)  := (l, wolne-1);
                    *[kanał(i) <> O ->  i  :=  i + 1];
                    (odbiór(m), kanał(i))  :=  (true, m);
                    [(k:l..M) k = m ->  ODBIORCA(k)!i];
                    NADAWCA(j)!i;
          ]
        []  (j:l..N)  NADAWCA(j)?ZWALNIAM(i)  ->
            (odbiór(kanał(i)), kanał(i), wolne)  :=
                                          (false,  O,  wolne+1);
      ]
  ]
```

W tym rozwiązaniu wybór wolnego kanału zaczyna się zawsze od numeru 1. W ten sposób wykorzystanie kanałów będzie nierównomierne — te o niższych numerach będą używane częściej niż te o wyższych numerach.
Można tego uniknąć wybierając kanał niedeterministycznie w następujący sposób:

```
[(k:l..K)  kanał(k) <> O -> i := k].
```

5.4.9 Obliczanie iloczynu skalarnego

Proces ZWIĘKSZ wykonuje N-krotnie zwiększanie i wysyłanie do procesu MNÓŻ wartości i, po czym kończy się. W procesach MNÓŻ i DODAJ pętle są nieograniczone, co może sugerować, że procesy te będą wykonywać się w nieskończoność. Jednak zgodnie z semantyką CSP, jeśli proces, na który czeka operacja wejścia w dozorze, nie istnieje (np. już się zakończył), to dozór jest fałszywy. Tak więc po otrzymaniu ostatniego i z procesu ZWIĘKSZ zakończy się proces MNÓŻ, co spowoduje, że zakończy się także proces DODAJ.
Schemat komunikacji między procesami przedstawia rys. 5.24.



Rys. 5.24. Obliczanie iloczynu skalarnego

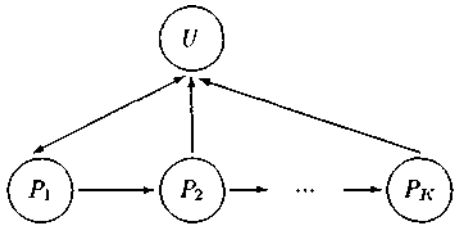
```
[ZWIĘKSZ::  i:  integer;
       i := 0;
       *[ i < N  ->  i := i  +  1;  MNÓŻ!i]
    ||  MNÓŻ:: i:  integer;  a,b:  (1..N)  real;
       *[ ZWIĘKSZ?! -> DODAJ!a(i)*b(i)]
    ||  DODAJ:: s.iloczyn:  real;
       iloczyn  := 0;
       *[ MNÓŻ?s  -> iloczyn  :=  iloczyn + s]
       WYJŚCIE!iloczyn
    ]
```

5.4.10 Obliczanie współczynników rozwinięcia dwumianu Newtona (n + b)n

Obliczanie wykonuje się „falami". Każda fala rozpoczyna się w P(l) i dociera o jeden proces dalej niż poprzednia. Liczba fal przechodzących przez dany proces jest o jeden mniejsza od liczby fal przechodzących przez proces poprzedni. Po przejściu odpowiedniej liczby fal procesy wysyłają obliczone współczynniki do procesu U. Zmienna n w procesie P(i), i = l,...,K, oznacza liczbę fal, które jeszcze mają przejść przez proces, natomiast x jest wartością i-tego elementu obliczanego właśnie wiersza trójkąta Pascala.

Przejście kolejnej fali polega na tym, że każdy proces dodaje do swojej starej wartości x starą wartość otrzymaną od procesu o numerze o jeden niższym, czyli y. W efekcie nowe wartości w poszczególnych procesach tworzą kolejny wiersz trójkąta Pascala. Zauważmy, że jeśli proces U wyśle do P(l) liczbę ujemną, to instrukcja alternatywy w procesie P(l) zakończy się z błędem, w rezultacie cały proces zakończy się z błędem. Procesy P(i), i = 2,...,K, wówczas zakończą się (poprawnie) jeden po drugim, ponieważ instrukcje wejścia w dozorze zwrócą wartość false.

Schemat komunikacji między procesami przedstawia rys. 5.25.



Rys. 5.25. Obliczanie współczynników dwumianu Newtona

```
comment K - liczba procesów liczących;
 [U:: i, n: integer;  w:  (1..K) integer;

   P(l)!n;
   i := 1;
   *[(j:l..K) j  =  i;  i <= n + l; P(j)?w(j)  ->  i  :- i + 1]
 ||P(1):: n: integer;
   *[ U?n ->
      C n = O -> skip
      [] n > O  ->  P(2)!n-l;
                  *[n >  l -> P(2)!l; n  := n -  1]
      ];
      U!l
    ]
 l |P(i:2..K)::  n:  integer; x, y: integer;
   *[P(i-l)?n -> x := 1;
      [n = O -> y  := O
      [] n > O  ->  P(i+l)!n-l; P(i-l)?y
      ];
      *[n>l ->  x := x + y; P(i+l)!x; n  := n -  1;  P(i-l)?y];
      U!x+y
    ]
 ]
```

5.4.11 Mnożenie macierzy przez wektor

Proces Y(i), i = l,...,N, sumuje kolejne iloczyny liczb x(j) i a(i,j), j = 1,... ,M, otrzymywane odpowiednio od Y(i-l) i A(i). Liczby x(j) są przekazywane dalej do procesu Y(i+l), i = 1, ... ,N-1. Procesy Y(0) oraz A(i), i = 1,... ,N najpierw pobierają, cały wektor od procesu U, a następnie wysyłają go do odpowiedniego procesu Y.

```
comment N,M  -  wymiary macierzy;
[U:: j, k: integer;  y: (1..N) real;  x:  (1..M) real;
          a:  (1..N,  1..M) real;

    j  :=  1;  *[j  <= M -> Y(0)!x(j);  j:=j  + 1];
    k  :=  1;
    *[(i:l..N) i = k -> j := 1;
                        *[j <= M -> A(i)!a(i,j), j  := j +  1];
                        k := k + l
     ];

    k  :=  1;   *[(!:!..N) i = k -> Y(i)?y(i); k := k +  1]
||A(i:l..N)::  a:  (1..M) real; j: integer;
     *[true  -> j  := 1; *[j <= M; U?a(j)  -> j := j + 1];
                 j  := 1; *[j <= M ->  Y(i)!a(j); j := j  +  1]
       ]
||Y(0)::  x:  (1..M) real; j: integer;
     *[true  -> j  := 1; *[j <= M ->  U?x(j); j := j + 1] ;
                 j  := 1; *[j <= M ->  Y(l)!x(j); j := j  +  1]
||Y(i:l..N)::  a,  x,  y: real; j: integer;
     *[true  -> (y, j)  := (O, 1);
          *[j  <= M -> Y(i-l)?x; A(i)?a;  y  := y + a *  x;
                       [i O N -> Y(i+l)!x
                       [] i = N -> skip] ;
                       j  := j + 1;
           ];
          U!y
     ]
]
```

Można zrezygnować z procesów pośredniczących Y(0) oraz A(l..N),jeśli proces użytkownika U będzie wysyłał dane w następującej kolejności. Najpierw x(l) i a(l,l) doY(l), następnie x(2) i a(l,2) doY(l) oraz a(2,l) do Y(2), potem x(3) i a(l,3) do Y(l), a(2,2) do Y(2) oraz a(3,l) do Y(3), itd. Na końcu powinien przekazać a(N,M) procesowi Y(N).

5.4.12 Obliczanie wartości wielomianu

Rozwiązanie wersji 1

Każdy proces P (i) zatrzymuje u siebie pierwszy otrzymany współczynnik (będzie to a(M+l-i)), a następne przekazuje procesowi P(i+l). W trakcie obliczeń proces P (i), i>l, otrzymuje od procesu P(i-l) oprócz argumentu x także sumę częściową a(M)*xx~2 + a(M-l)*x1~3 + ... + a(M+2-i), którą (zgodnie z algorytmem Homera) mnoży przez x, a następnie dodaje swój współczynnik a(M-H-i). Zmienna iksy wskazuje, z którym strumieniem danych mamy do czynienia, a zmienna mam — czy proces dostał już swój współczynnik. Jeśli do procesu nie dotarł współczynnik, a dotarł już argument, to otrzymywane następnie sumy częściowe przekazuje dalej bez zmian (są to po prostu wyniki końcowe obliczone przez

wcześniejsze procesy). Proces P(N) przekazuje procesowi użytkownika P(N+1) jedynie obliczone wyniki. Sygnał KONIEC() oddziela od siebie wyniki dla każdego wielomianu.

```
comment N - liczba procesów liczących;
[P(i:l..N):: iksy, mam: boolean; a, b, x, y: real;
  (iksy, mam) := (false, false);
  *[ not iksy; P(i-l)?b ->
                          [mam -> P(i+l)!b
                          [] not mam -> (mam, a) := (true, b)]
  [] iksy; P(i-l)?x  ->   [i <> l -> P(i-l)?y
                          [] i = l -> y := O ];
                          [i O N -> P(i+l)!x
                          [] i = N -> skip ] ;
                          [mam -> P(i+l)!y*x+a
                          [] not mam -> P(i+l)!y]
  [] P(i-l)? KONIECO ->   [i <> N -> P(i+l) !KONIEC()
                            [] i = N; iksy -> P(N+l)!KONIEC()
                            [] i = N; not iksy -> skip
                            ];
                            iksy := not iksy;
                            mam := mam and iksy;
  ]
]
```

Rozwiązanie wersji 2

Każdy proces P (i) zatrzymuje u siebie pierwszy otrzymany współczynnik, ale jeśli otrzyma następny, to ten, który już miał, przesyła procesowi P(i+l), i a zatrzymuje u siebie nowo otrzymany współczynnik (ostatecznie będzie to także a(M+l-i)). W trakcie obliczeń proces P(i), i > l, otrzymuje od procesu P(i-l) parę: argument i sumę częściową a(M)*x1-2 + a(M-l)*xi-3 + ... + a(M+2-i). (Proces P(l) zamiast sumy częściowej otrzymuje 0.)

Proces P(i) przesyła do procesu P(i+l) otrzymany argument oraz sumę częściową pomnożoną przez x z dodanym współczynnikiem a(M+l-i). Jak poprzednio, zmienna mam wskazuje, czy proces ma swój współczynnik. W tym przypadku proces nie musi pamiętać, z którym strumieniem danych ma do czynienia, wskazuje na to bowiem typ otrzymywanego komunikatu. Jeśli otrzyma pojedynczą liczbę, jest to na pewno współczynnik, jeśli parę liczb, jest to argument z sumą częściową. Sygnał KONIEC() informuje proces o tym, że skończyły się obliczenia dla danego wielomianu i że teraz albo otrzyma jakiś współczynnik z nowego wielomianu, albo będzie jedynie przekazywał wyniki obliczone przez wcześniejsze procesy.

```
comment  N - liczba procesów liczących;
[P(i:l..N):: wsp, mam: boolean; a, b, x, y:  real;
  mam  := false,
  *[P(i-l)?b ->  [ mam ->  P(i+l)!a
                 [] not mam -> mam  := true
                 ]
                 a := b;
  [] P(i-l)?(x,y) -> [mam  ->  P(i-H)! (x,y*x+a)
                      [] not mam -> P(i-H) ! (x,y)]
  [] P(i-l)?KONIEC() -> mam := false;
                        P(i+l)!KONIEC()
  ]
]
```

5.4.13 Mnożenie wielomianów

ad.1. Trzy zmienne logiczne mam_a, mam_b i mam_c służą do pamiętania, czy proces P(i,j) dostał już współczynnik odpowiednio a(j) i b(i) oraz wynik częściowy c. Dzięki użyciu tych zmiennych uzyskuje się większą wspólbieżność wykonywania procesów, nie można bowiem z góry przewidzieć, którą ze spodziewanych trzech wartości proces otrzyma jako pierwszą.

```
comment  N - maksymalny  stopień mnożonych wielomianów
[P(i:l..N-l,j:l..N-l)::  a,b,c: real;
     mam_a,  mam_b, mam_c: boolean;
     (mam_a, mam_b, mam_c)  := (false, false,  false);
     *[not mam_a; P(i-l,j)?a -> P(i+l,j)!a;  mam_a := true
      []  not mam_b; P(i,j+l)?b -> P(i,j-l)!b; mam_b := true
      []  not mam_c; P(i-l,j+l)?c -> mam_c  := true
      []  mam_a; mam_b; mam_c -> P(i+l,j-1)!c+a*b;
             (mam_a, mam_b, mam_c)  := (false, false, false)
     ]
]
```

Zauważmy, że przy takiej treści procesów P (i, j) jest możliwe potokowe obliczanie iloczynów dla wielu par wielomianów jednocześnie.

ad. 2. Procesy P (O,j), j = 0,...,N będą miały w pierwszym dozorze zamiast instrukcji P(i-l, j)?a instrukcję U?a. Analogicznie procesy P(i,N), i = O,...,N będą miały w drugim dozorze zamiast instrukcji P(i,j*l)?b instrukcję U?b. Procesy obu tych grup nie będą miały w ogóle zmiennej mam_c i trzeciego dozoru, a zamiast wartości c+a*b będą wysyłać a*b.
W procesach P(i,0),i = 0,...,N, nie będzie instrukcji P(i,j-l) !b, a w procesach P (N, j), j = O, . . . ,N, nie będzie instrukcji P(i+l,j) !a. Procesy obu tych grup będą miały zamiast instrukcji P(i+l, j-1) !c+a*b instrukcję U!c+a*b.

ad. 3. W powyższym rozwiązaniu obliczenia współczynników iloczynu dwóch wielomianów zaczynają się od prawego górnego rogu tablicy procesów P(0. .N,0..N). Zatem pierwszym procesem, który otrzyma oba współczynniki powinien być proces P (O, N), następnymi — procesy P(0,N-1) oraz P(1,N) itd., a ostatnimi — procesy P(0,0) i P(N,N), które od razu zwracają użytkownikowi iloczyn otrzymanych liczb. Użytkownik powinien więc przekazywać współczynniki w następującej kolejności a(N), b(0), a(N-l), b(l),a(N-2),b(2),...,a(0),b(N).

5.4.14 Sito Eratostenesa

Proces SITO(O) wysyła do procesu WYJŚCIE liczbę 2, a liczby nieparzyste z przedziału <3,N> wysyła do procesu SITO(l). Każdy z procesów SITO (l. .M) pierwszą liczbę, którą odbierze od poprzednika, wysyła do procesu WYJŚCIE, a wszystkie następne przesiewa i dalej posyła tylko te, które nie są podzielne przez pierwszą otrzymaną. Ostatni proces wysyła przesiane liczby bezpośrednio do procesu WYJŚCIE (powinny to być tylko liczby pierwsze).

```
comment M - liczba sit,
        N - górna  granica generowanych liczb  pierwszych;
 [SITO(O)::  i: integer;
```

```
      WYJŚCIE!2;
      i  := 3;  *[i <= N -> SITO(l)!i; i  :=  i + 2]
 ||SITO(j:l..M):: i, k: integer;
      SITO(j-l)?i;
      WYJŚCIE!i;
      *[SITO(j-l)?k ->
              [k mod i = O ->  skip
              [] k mod i <> O  ->  [j  = M -> WYJŚCIE!k
                                   []  j <> M -> SITO(j + l)!k]
             ]
      ]
]
```

Aby odpowiedzieć na pytanie, jaka jest minimalna liczba sit potrzebna do przesiania N liczb naturalnych, rozważmy liczbę pierwszą p, która trafi do ostatniego sita SITO (M). Niech q będzie następną liczbą pierwszą po p. Najmniejszą liczbą złożoną, którą przepuści SITO (M) jest q*q. Wszystkie mniejsze liczby złożone muszą bowiem być podzielne przez jakąś liczbę pierwszą mniejszą od q, a więc zostaną odsiane albo przez SITO (M), albo przez poprzednie sita.

Zatem do przesiania wszystkich liczb z zakresu <2,N>, potrzeba tyle sit, by N<q*q. Na przykład dla N=100 wystarczy M=3, gdyż do sit trafią liczby 3, 5, 7 a najmniejszą liczbą złożoną, którą przepuści SITO(3) jest 121 = 11 * 11. Dla N=1000 potrzeba tylko M=10 sit, w których ugrzęzną liczby 3, 5, 7, 11, 13, 17, 19, 23, 29, 31.

5.4.15 Sortowanie oscylacyjne

W procesie SORT użyto tablicy logicznej daj (zainicjowanej na true), której i-ty element otrzymuje wartość f alse wtedy, gdy proces ten odbierze wynik od procesu A(i), i = l,...,N. Dzięki temu posortowanych liczb nie odbiera się w kolejności indeksów lecz wtedy, gdy procesy A są gotowe do ich wysłania. Warto zwrócić uwagę na instrukcje równolegle w procesach A i B, które umożliwiają procesom komunikowanie się wtedy, gdy tylko są na to gotowe. Nie jest bowiem istotne, w jakiej kolejności procesy A komunikują się z odpowiednimi procesami B i odwrotnie. Warto też zwrócić uwagę na niedeterminizm instrukcji alternatywy w procesach A i B wówczas, gdy obie otrzymane liczby są równe.

Pętla w procesach A i B powinna wykonać się tyle razy, aby każda liczba mogła znaleźć się we właściwym procesie A. W najgorszym przypadku liczba musi „przewędrować" przez wszystkie procesy, a więc pętla musi wykonać się N razy.

```
comment N -  liczba liczb do posortowania;
 [SORT:: q:  (1..N) integer;  i: integer;
         daj:  (1..N)  boolean;
      i  := 1;
      *[(j:l..N) i =  j -> A(j)!q(j); i := i +  1];

      *[(!:!..N) not  daj(i)  -> daj(i) :=  true];
      *[(!:!..N) daj(i);  A(i)?q(i)  -> daj(i)  := false]
 || A(k:l..N):: i, ql, q2:  integer;
     *[ SORT?ql ->
         (q2,  i) :=  (ql,  N);
         *[  i  > O ->  [ql  >= q2 ->  [B(k-l)!q2  M  B(k)!ql]
                        []  q2 >= ql ->  [B(k)!q2  ||  B(k-l)!ql]
                        ];                                          i
                                                                   j
```

```
                    [B(k-l)?ql  l l B(k)?q2];
                    i := i - 1;
          ];
          SORT!ql
      ]
|| B(0)::  q:  integer; *[A(l)?q  -> A(l)!q]
|| B(N)::  q:  integer; *[A(N)?q  -> A(N)!q]
|| B(k:l..N-l)::  ql,  q2: integer;
     *[true  ->  [A(k)?ql  || A(k+l.)?q2] ;
                 [ql <= q2 ->  [A(k)!ql  ||  A(k+l)!q2]
                 [] q2 <= ql ->  [A(k-H)!ql ||  A(k)!q2]
                 ]
     ]
]
```

5.4.16 Tablica sortująca

W następującym rozwiązaniu wprowadzono dodatkowo sygnał DAJ(), który proces T(i) wysyła do procesu T(i+l), jeśli chce otrzymać od niego liczbę.

Każdy proces T(i) ma lokalną zmienną n wskazującą, ile liczb już otrzymał.

Wartość n-1 mówi, ile liczb proces przekazał już swemu następnikowi. Tylko tyle może potem od niego odebrać. Zauważmy, że jedynym zadaniem procesu T(N) jest pobieranie liczby od procesn T(N-l), a następnie ponowne jej zwracanie.

```
comment N  -  pojemność tablicy;
[T[i:l..N-l)::  a, x:  real; n:  integer;
  n  :=  0;
  *[ n=0;  T(i-l)?a -> n  := l
   [] n >  0;  n <= N - i; T(i-l)?x  ->
                            [a  > x  ->  T(i+l)!a; a  :=  x
                            []  a <=  x -> T(i+l)!x
                           ];
                           n  := n  + l
  [] n>0;  T(i-l)?DAJ()  ->  T(i-l)!a;
                           n  := n  - 1;
                            [n  = O  -> skip
                            []  n >  O ->  T(i+l)!DAJ();
                                           T(i+l)?a
                           ]
  ]
|| T(N)::  a:  real;
  *[T(N-l)?a -> T(N-1)?DAJ();  T(N-l)!a]
|| T(0)::  x:  (1..N)  real;
           i:  integer;

   i  := 1;
   *[i <= N ->  T(l)!x(i);  i  := i + 1] ;

   i  := 1;
   *[i <= N ->  T(1)!DAJ(); T(l)?x(i);  i := i +  1];

]
```

5.4.17 Porównywanie ze wzorcem

Algorytmy procesów U i P (i) są szczegółowo opisane w zadaniu. W procesie Z trzeba zaprojektować struktury danych. Na pierwszy rzut oka wydaje się, że należy użyć tablicy o długości M, dopasowanie wzorca bowiem może być rozpoznane na każdej z M pozycji tablicy b. Po dokładniejszym przyjrzeniu się można stwierdzić, że w danej chwili do procesu Z mogą trafiać jedynie liczby z ograniczonego zakresu. Jeśli proces P(I) rozpoznaje dopasowanie na pozycji n wysyłając do procesu Z liczbę n, to w tym samym czasie proces P (N) może w najgorszym razie, po wcześniejszym stwierdzeniu dopasowania także na pozycji n, próbować wysłać do procesu Z liczbę n-N+1. Zatem maksymalny zakres obejmuje n-(n-N+l) + l = N liczb. Z może więc pamiętać tylko N liczników. Musi jednak w jakiś sposób umieć, rozpoznać, że dana liczba przyszła do niego po raz pierwszy. Wydaje się, że w tym celu wystarczy pamiętać maksymalną otrzymaną liczbę, gdyż każda większa będzie na pewno nowa. Jednak ponieważ procesy P komunikują się z procesem Z w nieznanej nam kolejności, może się zdarzyć, że do Z dotrze najpierw liczba większa, a potem mniejsza (np. najpierw n od P(l), a potem n-N+1 od P(N), oczywiście, gdy N > 1). W związku z tym w procesie Z oprócz tablicy liczników licz zadeklarujemy tablicę widzianych liczb widz, na podstawie której będziemy inicjować liczniki. Tablicy tej nadamy wartości początkowe - N, ponieważ tej liczby nie wyśle na pewno żaden z procesów P.

```
 comment N  -  długość wzorca,
        M  -  długość tekstu;
 [U::  a: (1..N)  char; b:  (1..M)  char;  j: integer;

   j  := 1;
   *[(i:l..N)  i  = j -> P(i)!a(i); j  := j +  1];
   j  := D
  *[  j <= M  -> P(N)!b(j);  j  := j +  1]
|| P(i:  1..N):: a, b:  char; n:  integer;
    U?a;  n:  = 1;
    *[n <= M ->  [i  =  N -> U?b
                  []  i  O N -> P(i+l)?b] ;
                  [i  O l -> P(i-l)!b
                  []  i  = l -> skip] ;
                  [b  O a -> skip
                  []  b  = a -> Z!n-i+l] ;
                  n  :=  n + 1;
    ]
|| Z::  m,  j:  integer;  licz, widz:  (O..N-1)  integer;
    j  := 0;
    *[ j  <= N -  l  ->  widz(j)  :=  -  N; j  :=  j  + 1];
    *[(i:l..N) P(i)?m -> j  :=  m  mod N;
           C m O  widz(j) ->  (widz(j),  licz(j)) := (m,  1)
           [] m  =  widz(j) ->  licz(j)  := licz(j) + l
                               Clicz(j)  = N  -> WYJŚCIElm
                               []  licz(j) <> N -> skip]
           ]
    ]
]
```

5.5 Symetryczne spotkania w innych językach

5.5.1 occam

Język occam [Jone 87, Inmo88, Burn88, GallOO] został zaprojektowany z przeznaczeniem do programowania transputcrów, równoległych mikroprocesorów nowej generacji, w których wspólbieżność i synchronizacja są wspierane sprzętowo.
Pierwsza wersja języka, znana jako occam I powstała w rokri 1982. Druga — occam 2 — pojawiła się w roku 1986. Język, jak wskazuje jego nazwa, zaprojektowano zgodnie z zasadą brzytwy Ockhama, która głosi, że „bitów nie należy mnożyć bez konieczności".

Każda instrukcja w języku occam jest traktowana jako proces. Wyróżniono pięć instrukcji (procesów) podstawowych. Są to instrukcje: przypisania, wejścia, wyjścia, SKIP (instrukcja pusta) i STOP (odpowiadająca wiecznemu zapętleniu się programu).

W occamie, w odróżnieniu od CSP, dla każdej pary komunikujących się procesów trzeba zadeklarować jednokierunkowy kanał. W instrukcjach wejścia-wyjścia zamiast identyfikatorów procesów podaje się identyfikator kanału. W zależności od sposobu zadeklarowania, przez kanały można przesyłać albo pojedyncze wartości, albo tablice wartości określonego typu.

Procesy złożone tworzy się z procesów podstawowych i innych procesów złożonych za pomocą konstrukcji, z których trzy najważniejsze to konstrukcje: wykonania sekwencyjnego SEQ, wykonania równoległego P Wi wykonania alternatywnego ALT.

W occamie nie ma nawiasów początku i końca instrukcji złożonej, a strukturę programu zaznacza się przez odpowiednie wcięcia w tekście. Procesy tego samego poziomu mają takie samo wcięcie.

Sekwencję instrukcji w CSP postaci II; 12; 13 w occamie zapisuje się za pomocą konstrukcji SEQ:

```
SEQ
   11
   12
   13
```

Instrukcję wykonania równoległego w CSP postaci [P1 || P2 || P3] w occamie zapisuje się za pomocą konstrukcji PAR:

```
PAR
   Pl
   P2
   P3
```

Konstrukcja ALT jest odpowiednikiem instrukcji alternatywy z CSP. Na przykład instrukcję alternatywy w CSP postaci

```
[mam;  A?x ->  I1
[] mam;  B?y ->  I2
[] not  mam ->  I3
]
```

w occamie zapisuje się następująco:

```
ALT
   mam  & CA?x
     I1
   mam  & CB?y
     I2
   not  mam & SKIP
     I3
```

CA i CB są tutaj nazwami kanałów, przez które dany proces komunikuje się odpowiednio z procesem A i B. Instrukcja SKIP użyta w miejscu instrukcji wejścia oznacza, że do spełnienia dozoru wystarczy spełnienie warunku logicznego.

Konstrukcje: warunkowa IF, wyboru CASE i powtórzenia WHILE służą do tworzenia occamowych odpowiedników instrukcji warunkowej, wyboru i pętli. Warto zwrócić uwagę, że konstrukcja postaci

```
IF
  Wl
    I1
  W2
    I2
```

nie jest równoważna instrukcji alternatywy z CSP postaci

```
[Wl -> I1  []  W2  -> I2],
```

jeśli warunki W1 i W2 wzajemnie się nie wykluczają. W odróżnieniu od CSP, gdy jest spełniony więcej niż jeden warunek, do wykonania wybiera wsię zawsze instrukcję odpowiadającą pierwszemu spełnionemu warunkowi.

(Niedeterminizm można uzyskać stosując konstrukcję ALT z instrukcją SKIP w dozorze.)

Instrukcję pętli w CSP postaci

```
*[W -> I]
```

w occamie zapisuje się następująco:

```
WHILE W
  I
```

Jeśli pętla w CSP ma więcej niż jeden dozór, to tłumacząc ją na occain trzeba użyć kombinacji konstrukcji WHILE i ALT.

Efekt sparametryzowanych procesów i sparametryzowanych instrukcji dozorowanych uzyskuje się w occamie za pomocą mechanizmu powielania uzyskiwanego przez dopisanie do konstrukcji SEQ, PAR, ALT lub IF klauzuli postaci

```
indeks =  start FOR licznik
```

W konstrukcji SEQ zastosowanie powielania daje taki sam wynik, jak instrukcja pascalowa

```
for indeks := start to start+licznik-1.
```

Tablicę procesów w CSP postaci [P(i:1..N): : I(i)] w occamie zapisuje się następująco:

```
PAR i =  1  FOR N
  I(i)
```

Sparametryzowaną instrukcję dozorowaną w CSP postaci

```
[(i:l..N)  W(i) ->  I(i)]
```

w occamie można zapisać następująco:

```
ALT i =  l  FOR N
  W(i)
  I(i)
```

W języku occam 2 jest dostępnych siedem typów podstawowych: BYTE, i INT16, INT32, INT64 (integer wskazanej długości), REAL32, REAL64 i BOOL (boolean). Zmienne ł kanały deklaruje się podając typ i nazwę, np. BOOL b jest deklaracją zmiennej b typu logicznego, CHAN REAL32 c jest deklaracją kanału c, przez który można przesyłać wartości typu rzeczywistego. Tablice deklaruje się podając w nawiasach kwadratowych przed nazwą typu liczbę jej elementów, np. [10] INT16 a jest deklaracją dziesięciu zmiennych całkowitych o nazwach od a(0) do a(9), [10] CHAN BYTE c jest deklaracją tablicy dziesięciu kanałów, a CHAN [10] BYTE c jest deklaracją jednego kanału, przez który można przesyłać dziesięcioelementowe tablice.

Jak widać, zamieszczone w tym rozdziale rozwiązania w CSP można prawie automatycznie przetłumaczyć na occam. Jednak occam zawiera także mechanizmy niedostępne w CSP, takie jak procedury i funkcje. Przydatna, zwłaszcza przy programowaniu systemów czasu rzeczywistego, jest możliwość nadawania procesom priorytetów i korzystania z zegara czasu rzeczywistego. Occam umożliwia również konfigurowanie programu, czyli wskazywanie, na których procesorach mają być wykonywane poszczególne procesy.

Na maszynach typu IBM PC język occam jest dostępny dzięki karcie Inmos IMS B004, która wraz z systemem TDS (Transputer Devclopment System} zawierającym m.in. specjalny edytor, kompilator, debugger i biblioteki z wieloma procedurami i funkcjami, głównie do obliczeń numerycznych i wejścia-wyjścia, stanowi środowisko do uruchamiania programów w occamie.

5.5.2 Parallel C

Język Parallel C powstał w roku 1984 jako rozszerzenie standardowego języka C [KeR,i87] o mechanizmy programowania współbieżnego [ParaOl].

Rozszerzenie to polega na dodaniu dwóch instrukcji: par i select oraz jednego typu channel. Ponieważ Parallel C jest przeznaczony głównie do programowania transputerów, zastosowane konstrukcje językowe przypominają analogiczne mechanizmy w occamie.

Parallel C ma jednak wszystkie własności języka C, czego konsekwencją jest z jednej strony większa swoboda w operowaniu typami, ale z drugiej brak kontroli poprawności użycia mechanizmów programowania współbieżnego oraz możliwość wielu efektów ubocznych.
Instrukcja par ma takie samo znaczenie jak konstrukcja PAR w occamie.

```
par { II; 12; ...;  In }
```

oznacza współbieżne wykonanie instrukcji II, 12, ... i In, którymi mogą być również instrukcje złożone. Mechanizm powielania ma identyczną postać jak stosowany w pętli for w języku C.

```
par (i = 0;  i < N;  i++)
```

oznacza to samo co PAR i=0 FOR N-1 w occamie. Jednakże w Parallel C indeks może być dowolnego typu (prócz struktur i elementów tablic), a więc można n p. uruchomić współbieżnie po jednym procesie (instrukcji) dla każdego elementu listy.

Oto przykład:

```
par (l  = lista; l != NULL;  l = l->nast) { ZróbCoś(l) } .
```

(W C napis for (;;) oznacza pętlę nieskończoną. W Parallel C na podobnej zasadzie można próbować uruchomić nieskończenie wiele procesów pisząc par (; ;), ale oczywiście zakończyć się to musi błędem wykonania.)

Współbieżnie uruchomione procesy mogą korzystać ze zmiennych globalnych, ale odbywa się to w sposób nie gwarantujący wzajemnego wykluczania.

Do synchronicznej komunikacji między procesami służą, podobnie jak w occamie, kanały deklarowane jako obiekty typu channel. W odróżnieniu od occamu można przez nie przekazywać obiekty dowolnego typu. Instrukcje wejścia i wyjścia wyraża się za pomocą instrukcji przypisania, w których występują nazwy kanału. Jeśli nazwa kanału stoi po lewej stronie instrukcji przypisania, mamy do czynienia z instrukcją wyjścia, jeśli po prawej - z instrukcją wejścia. Jeżeli np. C jest zadeklarowane jako channel C, to instrukcja C = x ma takie znaczenie jak C!x, a instrukcja y = C, takie jak C?y. Liczba bajtów wstawiana do kanału i pobierana z niego zależy od typu zmiennych x i y. (Zaleca się oczywiście, aby były to te same typy. Jeśli nie są, wielkość przekazywanego komunikatu określa proces, który później przybył na spotkanie.).

Możliwe jest także bezpośrednie przekazywanie z kanału do kanału (choć nie oznacza to jednoczesnej synchronizacji trzech procesów). Na przykład proces postaci

```
for (;;) C2 =  (int) Cl
```

jest przykładem jednoelementowego bufora. Zgodnie z zasadą języka C najpierw oblicza się prawą stronę instrukcji przypisania, co wymaga synchronizacji z procesem wysyłającym komunikat do kanału C1, a dopiero potem lewą stronę, co wymaga synchronizacji z procesem odbierającym z kanału C2. W tym przypadku jest potrzebne dodatkowe wskazanie typu komunikatu, gdyż nie wynika on z samej instrukcji przypisania.

Odstępstwo od zasady, że jeden kanał powinien być używany do jednokierunkowej komunikacji między parą procesów, nie jest wykrywane przez kompilator i może prowadzić do błędów wykonania.

Odpowiednikiem instrukcji dozorowanej z instrukcjami wejścia w dozorach jest w Parallel C instrukcja select. Jednak w odróżnieniu od CSP i occamu dozory w tej instrukcji są wyliczane w sposób deterministyczny w kolejności występowania. Składnia jest podobna do składni instrukcji switch w języku C. Każda gałąź instrukcji select rozpoczyna się od słowa kluczowego alt, po którym może wystąpić warunek logiczny poprzedzony słowem kluczowym cond, a następnie wskaźnik do kanału poprzedzony słowem kluczowym guard. Dozorem może być także tylko warunek lub tylko wskaźnik do kanału. Wyliczenie dozoru polega na wyliczeniu warunku logicznego i sprawdzeniu, czy jakiś proces czeka na wysianie komunikatu przez wskazany kanał. Sprawdzenie to nie jest związane z przesianiem komunikatu przez kanał.

Przesianie następuje dopiero wskutek wykonania odpowiedniej instrukcji przypisania. Dzięki takiemu rozwiązaniu proces, który stwierdzi, że w kanale oczekuje komunikat, nie musi od razu sam go odbierać, lecz może odebrać go później lub nawet zlecić odebranie innemu procesowi.

Mechanizm powielania można zastosować również wewnątrz instrukcji select. Przykładowo instrukcję w CSP postaci

```
C(i:0..N-l)  P(i)?x ->  I(i)
[]  Wl;  Q?y  -> II
```

```
 []  W2 -> 12
 []  R?z ->  skip]
```

można (z dokładnością do niedeterminizmii) zapisać w Parallel C następująco:

```
select {
   alt (i =  0; i <  N;  i++) guard &C[i] :
                              x  = C [i];  I(i) ;  break;
   alt cond  Wl guard &C1:  y = Cl; II; break;
   alt cond  W2: 12; break;
   alt guard &C2: z =  C2;  break;
}
```

przy czym channel C [N], Cl, C2 są kanałami, przez które dany proces komunikuje się z procesami odpowiednio P(0. .N-l), Q, R.

Jeżeli żaden z dozorów nie jest spełniony, bo odpowiednie kanały nie są jeszcze gotowe do transmisji, proces wykonujący instrukcję select zawiesza się w oczekiwaniu na gotowość jednego z nich. W Parallel C istnieje możliwość określenia maksymalnego czasu tego oczekiwania. Jeżeli posłowie select dopiszemy klauzulę within E, przy czym E jest wyrażeniem o wartości całkowitej, a jako ostatnią gałąź podamy alt timeout: I, to maksymalny czas oczekiwania będzie liczbą cykli zegara wskazaną przez wyrażenie E. Po tym czasie wykona się instrukcja I. Jeśli słowo timeout poprzedzimy warunkiem cond W, to czekanie zakończy się tylko wtedy, gdy warunek W będzie spełniony.

Zauważmy, że nie wszystkie przytoczone w tym rozdziale rozwiązania dadzą się prosto przetłumaczyć na Parallel C. Dotyczy to tych przypadków, w których świadomie zastosowano niedeterminizm do zapobiegania zagłodzeniu. Realizacja instrukcji select w Parallel C nie ma, własności żywotności. Możliwym rozwiązaniem tego problemu jest umieszczenie warunków i wskaźników do kanałów w cyklicznej liście, której przeglądanie w instrukcji select będzie za każdym razem zaczynać się od innego miejsca.

5.5.3 Edip

Język programowania współbieżnego Edip powstał w rokii 1988 w Instytucie Informatyki Uniwersytetu Warszawskiego [DuWy90a,DuWy90b]. Jest to wariant stworzonego przez P. Brinch Hansena języka Joycc [BrinSG].

Edip został zaprojektowany z myślą o systemach wielomikroprocesorowych (a więc i z myślą, o sieciach transpnterów). Autorzy starali się stworzyć możliwie prosty język umożliwiający programowanie współbieżne w sposób modularny. Notację Edipa oparto na notacji stosowanej w języku Edison [Brin83], a z CSP zapożyczono notację dla spotkań.

Procesy w Edipie komunikują się za pomocą synchronicznych komunikatów przesyłanych przez dwukierunkowe kanały. Procesy tworzą kanały dynamicznie za pomocą instrukcji form i identyfikują je przez specjalne zmienne typu portowego. Deklarując typ portowy specyfikuje się wszystkie komunikaty, jakie mogą być przekazywane przez kanał tego typu. Komunikat składa się z nazwy i typu danych, jakie zawiera. Komunikat złożony tylko z nazwy nazywa się syynalem.

Przykładowo port strumień(DANE(int) , WYNIKI(char)) jest deklaracją typu portowego, var KANAŁ: strumień jest deklaracją zmiennej typu portowego, form KANAŁ jest instrukcją utworzenia kanału, po której dopiero można z niego korzystać wykonując instrukcje wyjścia, np. KANAŁ !DANE(5), KANAŁ! WYNIKI ('a') i wejścia, np. KANAŁ?DANE(n), KANAL?WYNIKI(c), przy czym n jest typu int, a c jest typu char.

Procesy są tworzone i uruchamiane dynamicznie za pomocą instrukcji start. Proces może tworzyć i uruchamiać inne (zadeklarowane w nim) procesy, które działają z nim współbieżnie. Proces-ojciec może zakończyć się dopiero po zakończeniu wszystkich utworzonych przez niego procesów-synów.

Jest możliwe rekurencyjne tworzenie i uruchamianie procesów. Parametrami uruchamianych procesów mogą być zmienne typu portowego identyfikujące kanały, przez które proces może komunikować się ze światem zewnętrznym (zwłaszcza z ojcem). Kanał przestaje istnieć, gdy przestaje istnieć proces, który go utworzył.

Do jednego kanału może mieć dostęp wiele procesów, jednak w danej chwili tylko dwa z nich mogą się przez niego komunikować. Jeśli zatem proces wykonuje instrukcję wyjścia do kanału, przy którym czeka już wiele procesów pragnących wykonać instrukcję wejścia z tym samym typem komunikatu, to komunikat zostanie przekazany procesowi losowo wybranemu spośród czekających. Podobnie dokonuje się losowania spośród procesów czekających na wykonanie instrukcji wyjścia, jeśli jakiś proces zapragnie wykonać instrukcję wejścia.

W Edipie są dostępne trzy rodzaje instrukcji dozorowanych: warunkowa, pętli i ankietująca. Instrukcja warunkowa ma postać

```
if  Dl do SL1
else D2 do  SL2
else Dn do  SLn end
```

przy czym dozór Di jest tu warunkiem logicznym, a SLi jest listą instrukcji.

Dozory wylicza się po kolei i wykonuje się pierwszą listę instrukcji, której dozór jest spełniony. Nie ma tu więc niedeterminizmu.

Instrukcja pętli ma. identyczną postać z tym, że słowo kluczowe if jest zastąpione słowem while.

Instrukcja ankietująca ma w miejscu słowa if słowo poił. Każdy dozór instrukcji ankietującej jest postaci Bi & Ci, przy czym Bi jest wyrażeniem logicznym a Ci jest instrukcją wejścia lub wyjścia. Dozór jest spełniony, jeśli wyrażenie logiczne Bi ma wartość true i może być wykonana instrukcja wejścia-wyjścia Ci. Instrukcja ankietująca jest wykonywana tylko wtedy, gdy jest spełniony któryś z jej dozorów. Wykonanie polega wówczas na wyliczeniu wyrażenia Bi, wykonaniu instrukcji Ci, a następnie wykonaniu listy instrukcji SLi.

Język Edip jest dostępny na maszynach typu IBM PC. Kompilator tłumaczy tekst programu w Edipie na kod pośredni, który jest następnie interpretowany. Wspólbieżność jest tu realizowana na zasa.dzie podziału czasu procesora. Parametrem interpretatora jest liczba kwantów czasu procesora centralnego przydzielana każdemu procesowi z programu użytkownika.

6 Asymetryczne spotkania w Adzie

6.1 Wprowadzenie

6.1.1 Informacje ogólne

Język Ada powstał na zlecenie Departamentu Obrony Stanów Zjednoczonych jako język przeznaczony głównie do programowania systemów czasu rzeczywistego. Współbieżność jest nieodłączną cechą takich systemów, dlatego język Ada wyposażono w wiele udogodnień umożliwiających łatwe programowanie procesów współbieżnych. Autorzy Ady projektując konstrukcje językowe przeznaczone do programowania współbieżnego wykorzystali wiele pomysłów zaczerpniętych z CSP. Jest jednak także wiele różnic i te będziemy chcieli tutaj uwypuklić omawiając krótko mechanizmy Ady.

Ada jest językiem bardzo rozbudowanym o dość złożonej składni. Niestety nie udało się, jak dotąd, zrobić dobrego kompilatora pełnego języka na komputery typu IBM PC (najwięcej kłopotów sprawia właśnie realizacja współbieżności). Z tego powodu język ten nie jest u nas zbyt popularny. Istnieją jednak jego kompilatory na większe maszyny, a sama Ada staje się powoli standardowym językiem do zapisu programów współbieżnych w publikacjach naukowych, podobnie jak Pascal jest już od lat standardowym językiem do zapisu programów sekwencyjnych.

Czytelników bliżej zainteresowanych samym językiem odsyłamy do książek [Pyle86, HaPe89]. Mechanizm spotkań w Adzie jest także krótko omówiony w książce [BenA89].
W tym rozdziale podajemy tylko niezbędne informacje potrzebne do zrozumienia zamieszczonych dalej przykładów i rozwiązania proponowanych zadań. Mechanizm asymetrycznych spotkań został także wykorzystany w języku Concurrent C [GeRo88], którego tu nie omawiamy.

6.1.2 Deklaracje i instrukcje sterujące

Ada jest imperatywnym językiem programowania. Udostępnia cztery typy standardowe: Boolean, Integer, Float (który odpowiada pascalowemu real) i Character. Jest wiele innych typów prostych i strukturalnych, ale spośród nich będą nas interesować przede wszystkim tablice. Oto przykład deklaracji zmiennej typu tablicowego:

```
wektor:  array(1..10) of  Integer;
```

W chwili deklaracji można zmiennym nadawać wartości początkowe, np.

```
i: Integer := 0; w: array(1..5) of Integer:= (1,0,0,0,0).
```

W tym drugim przypadku mamy do czynienia z agregatem definiującym stalą typu tablicowego. W agregatach można użyć słowa kluczowego others oznaczającego wszystkie pozostałe wartości, zamiast zapisu (1,0,0,0,0) można więc napisać (1, others => 0).

Instrukcja warunkowego rozgałęzienia ma ogólną postać

```
if W1 then I1;
elsif W2 then  I2;
...
elsif Wn then  In;
else  I;
end if
```

przy czym W1, W2, ... Wn są warunkami, a I1,I2, ..., In, I ciągami instrukcji.
(Możliwe są postaci skrócone bez części elsif i/lub else.)
Warto w tym miejscu zauważyć, że oprócz operatorów logicznych or i and są także dostępne operatory or else i and then, które nie wymagają wyliczania drugiej części alternatywy (koniunkcji), jeśli pierwsza część okaże się prawdziwa (fałszywa).

W Adzie jest kilka rodzajów instrukcji pętli:

`loop I; end loop` jest pętlą nieskończoną, ale można z niej wyjść umieszczając wewnątrz ciągu instrukcji I instrukcję exit when W. W wyniku jej wykonania pętla zakończy się, jeśli warunek W będzie spełniony.

`for i in n..m  loop I; end loop` jest pętlą wykonywaną m-n+1 razy z wartością i zmieniającą się od n do m (oczywiście n < m); jeśli po słowie kluczowym in umieścimy słowo kluczowe reverse, to wartość i będzie się zmieniać w odwrotną stronę, tzn. od m do n (tu także n < m).

`while W loop  I;  end loop` jest pętlą wykonywaną tak długo, jak długo zachodzi warunek W.

6.1.3 Procedury i funkcje

Procedury w Adzie deklaruje się podobnie jak w Pascalu. Parametry procedury mogą być trojakiego rodzaju: wejściowe oznaczane słowem kluczowym in (mogą one występować tylko po prawej stronie instrukcji przypisania), wyjściowe oznaczane slowem kluczowym out (mogą występować tylko po lewej stronie instrukcji przypisania) oraz wejściowo-wyjściowe oznaczane słowem kluczowym inout. Oto przykład:

```
procedure  zwieksz(a:  inout Integer;  b:  in Integer)  is
begin
  a  : =  a + b;
end zwiększ;
```

Program główny w Adzie jest wyróżnioną procedurą.
Dla funkcji w nagłówku podaje się po słowie kluczowym return jej typ, a wartość funkcji jest zwracana instrukcją return.

```
function dodaj(a,b: in  Integer) return Integer  is
begin
  return a + b;
end zwiększ;
```

6.1.4 Pakiety

Pakiet jest zbiorem deklaracji danych i procedur, który można niezależnie kompilować, umieszczać w bibliotece oraz udostępniać innym pakietom i programowi głównemu. Instrukcje wejścia-wyjścia do komunikacji z użytkownikiem przez klawiaturę i ekran są zawarte w standardowym pakiecie Ady o nazwie Text_IO. Aby wewnątrz jakiejś procedury skorzystać z procedur innego pakietu, trzeba jej deklarację poprzedzić słowem kluczowym with, po którym następuje nazwa pakietu, np. with Text_IO.

Pakiety deklaruje się w dwóch częściach. W części specyfikacyjnej podaje się nagłówki procedur udostępnianych przez pakiet, np.

```
package czytelnia  is
  procedure czytanie(x: out Integer);
  procedure pisanie(x:  in Integer);
end;
```

w części implementacyjnej zaś treści udostępnianych procedur oraz deklaracje obiektów lokalnych i treść pakietu, np.

```
package body czytelnia  is

  procedure czytanie(x: out Integer)  is

  end czytanie;
   procedure  pisanie(x:  in Integer)  is

   end pisanie;
 end czytelnia;
```

(Pełną treść pakietu czytelnia podamy w przykładzie 6.2.3.)

6.1.5 Procesy i wejścia

Procesy,w Adzie oznacza się słowem kluczowym task. Można je deklarować wewnątrz procedury bądź pakietu. Z chwilą uruchomienia procedury równolegle z jej treścią wykonują się wszystkie zadeklarowane w niej procesy i procesy zadeklarowane w pakietach, z których ta procedura korzysta. (Nie ma więc potrzeby wprowadzania oddzielnego pojęcia instrukcji równoległej.)

Procesy w Adzie komunikują się za pomocą mechanizmu synchronicznych spotkań. Spotkania te różnią się jednak istotnie od spotkań w CSP. Podczas spotkania w Adzie mogą być dodatkowo wykonywane pewne obliczenia, przy czym mogą być one wykonywane tylko w jednym z komunikujących się procesów. Proces ten będziemy nazywać procesem obslugującym (server). Proces obsługujący udostępnia procesowi obslugiwanemu (client) różne miejsca spotkań, z którymi może wiązać różne obliczenia. Miejsca spotkań są widoczne na zewnątrz w postaci wejść (entries] do procesu obsługującego. Proces obsługiwany musi znać nazwę procesu obsługującego i wejście, przez które chce się z nim spotkać. Proces obsługujący natomiast nie zna nazwy procesu obsługiwanego. Spotkanie w Adzie jest więc asymetryczne zarówno ze względu na jednostronną identyfikację partnerów, jak i na jednostronną aktywność podczas spotkania. Zauważmy, że ten sam proces może byćjednocześnie procesem obsługującym (udostępniającym swe wejścia) i procesem obsługiwanym (korzystającym z wejść innych procesów).

Deklaracja procesu, podobnie jak deklaracja pakietu, składa się z dwóch części. W części specyfikacyjnej podaje się specyfikacje udostępnianych wejść (w przypadku procesów, które są tylko obsługiwane, część ta jest pusta, ale musi wystąpić), a w części implementacyjnej — treść procesu. Specyfikacja wejścia ma taką samą postać jak nagłówek procedury. Na przykład

```
task BUFOR is
  entry DAJ(x:  out Integer);
 entry WEZ(x:  in Integer);
end;
```

jest specyfikacją procesu BUFOR udostępniającego dwa wejścia: DAJ wymagające parametru wyjściowego i WEŹ wymagające parametru wejściowego.

6.1. WPROWADZENIE

Proces pragnący spotkać się z procesem BUFOR w wejściu DAJ, musi w swojej treści mieć instrukcję postaci BUFOR.DAJ(y), przy czym y musi być zmienną typu Integer.

Możliwe jest także wyspecyfikowanie tablicy wejść bezparametrowych.
Na przykład:

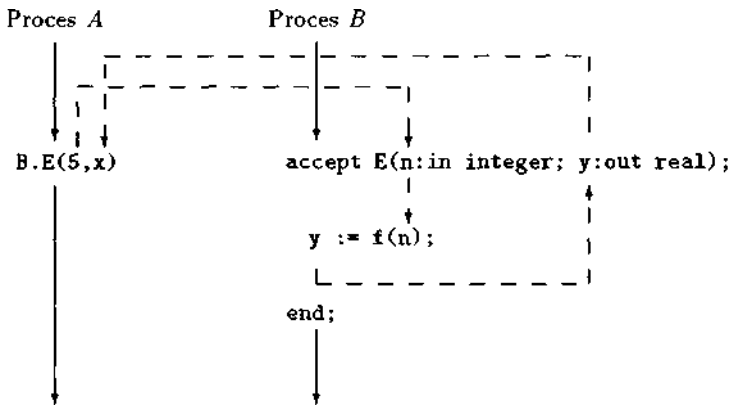
```
entry  E(1..5)
```

jest specyfikacją pięciu wejść: E(1) , E(2) , E(3) , E(4) i E(5).

6.1.6 Instrukcja accept

Jeżeli w części specyfikacyjnej procesu P podano wejście E, to w jego części implementacyjnej musi pojawić się co najmniej jedna instrukcja postaci
accept E do I end
slowo end można opuścić, jeśli I jest pojedynczą instrukcją, a jeśli I jest instrukcją pustą, można także opuścić słowo kluczowe do).

Ciąg instrukcji I jest wykonywany podczas spotkania przez proces obsługujący. Spotkanie wymaga synchronizacji procesów w sposób podobny jak w CSP. Proces wykonujący instrukcję P.E czeka, aż sterowanie w P dojdzie do instrukcji accept E, a proces P po dojściu do instrukcji accept E czeka, aż sterowanie w pewnym innym procesie dojdzie do instrukcji P.E. Gdy sterowania w obu procesach znajdą się w odpowiednich miejscach, następuje spotkanie. Jego przebieg jest jednak bardziej złożony niż w CSP. Najpierw z procesu obsługiwanego do procesu P są przekazywane wartości parametrów typu in i inout. Następnie w procesie P wykonuje się ciąg instrukcji I, po czym wartości parametrów typu out i inout są przekazywane z powrotem do procesu obsługiwanego. Przepływ danych między procesami ilustruje rys.
6.1.



RYS. 6.1. Asymetryczne spotkanie w Adzie

Jak więc. widać, w Adzie, w odróżnieniu od CSP, podczas jednego spotkania dane mogą być przekazywane między procesami w obie strony. Parę instrukcji B!x, A?y można zatem zrealizować w Adzie na dwa sposoby:

```
   proces A:  B.WEZ(x);
   proces B:  accept  WEZ(x:   in  Integer)  do y :=  x;  end WEŹ;
albo
   proces A:  accept  DAJ(y:   out  Integer)  do y := x;  end DAJ;
   proces B:  A.DAJ(y)
```

Ponieważ w Adzie proces obsługujący P nie wie, jakie procesy obsługuje, może się zdarzyć, że na spotkanie w jednym wejściu czeka niezależnie kilka procesów. W takiej sytuacji, w chwili dojścia sterowania do instrukcji accept proces P realizuje spotkanie tylko z pierwszym z czekających procesów. Pozostałe czekają dalej, aż P ponownie dojdzie do odpowiedniej instrukcji accept. Inaczej mówiąc, z każdym wejściem do procesu obsługującego jest związana kolejka procesów czekających na spotkanie w tym wejściu. Kolejka ta jest obsługiwana w kolejności zgłoszeń w miarę wykonywania odpowiednich instrukcji accept. Zauważmy, że z jednym wejściem może być związanych kilka instrukcji accept.

6.1.7 Instrukcja select

W CSP instrukcje wejścia mogą pełnić rolę dozorów w instrukcji alternatywy lub pętli. W Adzie uzyskano ten efekt wprowadzając specjalną instrukcję select. W procesie obsługującym ma ona następującą postać:

```
   select  11; or 12;  or  ...  or In;  else I; end select;
```

Każdy z ciągów instrukcji 11, ... , In musi zaczynać się od instrukcji accept być może poprzedzonej dozorem postaci when W =>.

Z uwagi na możliwość występowania klauzuli else semantyka tej instrukcji jest dość złożona. W pewnym uproszczeniu została ona opisana w książce [BenA89]. Szczegóły można znaleźć w [Pyle86] i [HaPe89]. Istotę działania instrukcji select wyjaśnimy porównując ją z instrukcją alternatywy w CSP.
Na przykład instrukcję postaci

```
    [ wl;w2;Pl?xl -> 11
    [] w3;  P2?x2  ->  12
    [] P3?x3 -> 13
```

```
]
```

w Adzie można zapisać następująco:

```
select
   when  wl  and then w2 =>
      accept WEZl(x: in  Integer)  do  xl := x;  end WEŹl;
      11;
or
  when w3 =>
      accept WEZ2(x: in  Integer) do  x2 := x;  end WEŹ2;
      12;
or
      accept WEZ3(x: in  Integer) do  x3 := x;  end WEŹ3;
      13;
end  select;
```

Odpowiednia gałąź instrukcji `select` jest wybierana w zależności od spełnienia dozorów `wl and w2` oraz `w3` (w gałęzi trzeciej dozorem jest domyślnie true) i możliwości spotkania w instrukcjach `accept WEŹ`.

Jeśli w żadnej gałęzi ze spełnionym dozorem nie może dojść do spotkania, to instrukcja select zawiesza się w oczekiwaniu na takie spotkanie. Jeśli żaden dozór nie jest spełniony, to wykonanie tej instrukcji kończy się błędem.

Jeśli w instrukcji select umieścimy na końcu jeszcze klauzulę `else I`, to ciąg instrukcji I wykonuje się wówczas, gdy wszystkie dozory są fałszywe lub gdy nie może dojść do spotkania w gałęziach ze spełnionymi dozorami.

Jeśli zamiast instrukcji I umieści się słowo kluczowe `terminate`, proces kończy się, gdy wcześniej zakończyły się wszystkie procesy spotykające się z nim w obrębie instrukcji select.

Jeżeli naszą przykładową instrukcję select umieścimy wewnątrz instrukcji loop, to uzyskamy ten sam rezultat co dopisanie gwiazdki w CSP. Jednak gdyby nasza instrukcja select składała się tylko z dwóch pierwszych gałęzi, to umieszczając ją w pętli trzeba by dopisać jeszcze klauzulę else postaci

```
else exit when  not (wl  and w2) and  then not  w3;
```

Zauważmy, że sama klauzula `else exit` tu nie wystarczy, gdyż wówczas pętla kończyłaby się także wtedy, gdy aktualnie nie jest możliwe żadne spotkanie.

Instrukcję `pxista` oznacza się słowem kluczowym `null`.

Instrukcja select może być również użyta w procesie obsługiwanym. Umożliwia ona wówczas sprawdzenie, czy procesy obsługujące mogą w danej chwili zrealizować spotkanie, a w razie odpowiedzi negatywnej pozwala na wykonanie ciągu instrukcji umieszczonego po else (por. przykład 6.3.1).

6.1.8 Atrybuty

Byty nazwane (typy, zmienne, procesy, itd.) w języku Ada mają pierwotnie zdefiniowane własności zwane atrybutami. Do atrybutów można odwoływać się pisząc nazwę bytu, apostrof i nazwę atrybutu. Spośród wielu atrybutów dostępnych w Adzie będziemy w tym rozdziale używać tylko jednego:

- count — atrybut wejścia wskazujący, ile aktualnie procesów czeka na spotkanie w tym wejściu.

W odróżnieniu od CSP, w Adzie nie ma możliwości odwołania się wewnątrz procesu do odpowiadającego mu indeksu w tablicy procesów. Inaczej mówiąc, proces nie może sam się zidentyfikować. Rozwiązanie tego problemu omawiamy w p. 6.1.9. Wymaga ono wprowadzenia w procesie dodatkowej zmiennej i instrukcji accept. Ponieważ trzeba by to zrobić w większości omawianych tu przykładów i rozwiązań, dla przejrzystości zapisu założymy, (podobnie, jak uczyniono to w [WeLi81]), że istnieje

- index — atrybut procesu w tablicy procesów, wskazujący wartość indeksu w tej tablicy (jest to więc coś w rodzaju numeru procesu).

6.1.9 Ograniczenia

W CSP istnieje możliwość wyspecyfikowania zarówno tablicy procesów, jak i tablicy dozorów, w których mogą wystąpić instrukcje wejścia dotyczące tychże procesów. W Adzie jest możliwość zadeklarowania tablicy procesów, można także wyspecyfikować tablicę wejść. Stablicowane wejścia muszą być co prawda bezparametrowe, służą więc jedynie do synchronizacji procesów, nie jest to jednak wielkie ograniczenie, gdyż po uprzedniej synchronizacji procesy mogą przesiać sobłe parametry za pomocą innego uzgodnionego wejścia.

Poważny problem wiąże się natomiast z brakiem możliwości stablicowania galęzi w instrukcji select. Oznacza to w praktyce, że jeśli proces obsługujący ma czekać na żądanie od dowolnego procesu z tablicy procesów, to w instrukcji select trzeba wypisać, jawnie wszystkie instrukcje accept. Czasami może to być bardzo kłopotliwe.

Problem ten można częściowo ominąć stosując pętlę typu for z instrukcją select tylko z jedną gałęzią, ale zmieniającą się po każdym wykonaniu pętli (por. przykład 6.2.4). Rozwiązanie to ma jednak dwie istotne wady. Po pierwsze, wymaga aktywnego czekania (nie wiadomo, które żądanie przyjdzie pierwsze, więc trzeba cyklicznie sprawdzać wszystkie). Po drugie, sprawdzanie odbywa się w ustalonej kolejności, nie ma więc niedeterminizmu.

Wspomnieliśmy wcześniej, że w Adzie nie ma możliwości odwołania się wewnątrz procesu do odpowiadającego mu indeksu w tablicy procesów. Inaczej mówiąc, proces nie wie, kim jest, i może się tego dowiedzieć dopiero, gdy skomunikuje się z procesem, w którym został utworzony. Tak więc każdy proces, który musi znać swój numer, powinien mieć zadeklarowane wejście, np. entry NUMER(nr: in Integer), oraz zmienną lokalną, np. moj_nr: Integer, a jedną z pierwszych instrukcji tego procesu powinna być instrukcja postaci

```
accept NUMER(nr: in Integer) do moj_nr := nr end NUMER;
```

Ponieważ jest to zabieg czysto techniczny, dla uproszczenia założymy istnienie dodatkowego atrybutu index (patrz p. 6.1.8).

6.2 Przykłady

6.2.1 Wzajemne wykluczanie

Pokażemy, jak w Adzie można zrealizować semafor binarny. Realizacją semafora ogólnego jest proces LOKAJ z przykładu 6.2.4. Realizacje semaforów innych rodzajów są przedmiotem zadań 6.3.1 i 6.3.2. Realizacja monitora w języku Ada jest przedmiotem zadania 6.3.3.

Ponieważ na semaforze binarnym można na przemian wykonywać operacje PB i VB, proces realizujący taki semafor może wyglądać nastepująco (zakładamy, że na początku semafor jest podniesiony):

```
task SEM_BIN is
   entry VB;
   entry PB;
end;

task body SEM_BIN  is
begin
   loop
    accept PB;
    accept VB;
   end  loop;
end SEM_BIN;
```

Przy takiej implementacji próby (niedozwolonego) wykonania operacji VB na podniesionym semaforze spowodują zawieszenie procesu. W istocie więc jest to realizacja semafora dwustronnie ograniczonego o wartości maksymalnej 1. Możemy jednak uzyskać efekt semafora binarnego wymagając, aby proces obsługiwany miał zamiast instrukcji SEM_BIN.VB instrukcję

```
select
  SEM_BIN.VB;
else
  błąd;
end select;
```

Gdy proces SEM_BIN czeka na wykonanie accept PB (co oznacza semafor podniesiony), próba wykonania SEM_BIN.VB nie powiedzie się i zostanie wywołana procedura błąd.

6.2.2 Producent i konsument

Komunikacja bez bufora

Przedstawiamy proces, który umożliwia przekazywanie porcji między dowolną liczbą producentów a dowolną liczbą konsumentów bez wykorzystania bufora. W procesie NAGANIACZ zastosowano zagnieżdżone instrukcje accept, dzięki czemu uczestniczy on jednocześnie w dwóch spotkaniach. Jedno odbywa się z jakimś producentem wywołującym wejście PROD, drugie z jakimś konsumentem wywołującym wejście KONS. Podczas tego trójstronnego spotkania proces NAGANIACZ przekazuje konsumentowi to, co wyprodukował producent. Typ porcja jest typem przekazywanych elementów, który można dobrać w zależności od potrzeb.

```
task NAGANIACZ is
   entry  PROD(pl: in  porcja);
   entry  KONS(p2: out porcja);
end NAGANIACZ;

task body NAGANIACZ  is
begin
   loop
```

```
       accept PROD(pl:  in porcja)  do
          accept KONS(p2: out  porcja) do
            p2 := pl;
          end KONS;
       end  PROD;
      end  loop;
   end;
```

Przykład ten zaczerpnięto z książki [LeVe85].

Komunikacja przez bufor N-elementowy

Oto przykład systemu złożonego z jednego producenta i jednego konsumenta komunikujących się za pomocą N-elementowego bufora. Poniższy tekst jest tłumaczeniem na Adę przykładu 5.2.3.

```
   task PRODUCENT is
   end;
   task body PRODUCENT  is
      p: porcja;
   begin
      loop
    produkuj(p);
    BUFOR.WEŹ(p);
  end loop;
end;

task KONSUMENT is
end;
task body KONSUMENT is;
  p:porcja;
begin
  loop
    BUFOR.DAJ(p);
    konsumuj(p)
  end loop;
end;

task BUFOR is
  entry DAJ(x: out porcja);
  entry WEZ(x: in porcja);
end;
task body BUFOR is
  N: constant Integer = ?;
  buf: array(O..N-l) of porcja;   — bufor N-elementowy
  d: Integer := 0;               -- miejsce do  wstawiania
  z: Integer := 0;               -- miejsce do  pobierania
begin
  loop
    select
      when d < z+N =>
        accept WEZ(x: in porcja)  do
          buf(d mod N) := x;
        end WEŹ;
        d := d + 1;
    or
      when z < d =>
        accept DAJ(x: out porcja) do
          x := buf(z mod N);
        end DAJ;
```

```
        z := z + 1;
    end select
  end loop;
end BUFOR;
```

Zauważmy, że części specyfikacyjne procesów PRODUCENT i KONSUMENT są puste, procesy te nie świadczą bowiem żadnych usług innym procesom. Jedynym procesem, który świadczy usługi jest BUFOR.

Nie potrzeba tu pomocniczego komunikatu JESZCZE(), ponieważ nie wymaga się, aby w dozorach występowały tylko instrukcje wejścia. Dzięki temu rozwiązanie w Adzie jest symetryczne ze względu na producenta i konsumenta, a przez to bardziej przejrzyste.

Gdy liczba wyprodukowanych porcji przekroczy maksymalną liczbę typu Integer,powstanie nadmiar w instrukcji dodawania d := d + 1. Aby tego uniknąć, można po niej dopisać instrukcję

```
if  d = k*N then  d  :=  N;  z  := z -  (k-l)*N; end if;
```

przy czym k jest odpowiednio dobraną liczbą całkowitą (np. maxint div N).

6.2.3 Czytelnicy i pisarze

Dwie wersje rozwiązania w Adzie problemu czytelników i pisarzy można znaleźć w książce [BenA89] (zad. 6.6). Tutaj pokażemy, w jaki sposób proces zarządzający dostępem do czytelni można umieścić w pakiecie. Pakiet czytelnia udostępnia na zewnątrz procedury czytanie i pisanie, sposób ich synchronizacji jest ukryty przed użytkownikiem.

Rozwiązanie z możliwością zagłodzenia pisarzy

```
package czytelnia  is
  procedure  czytanie(x:  out  Integer);
  procedure  pisanie(x:  in  Integer);
end;

package body czytelnia is
  książka: Integer;
  task PORTIER is
     entry ZACZYNAM;
     entry KOŃCZĘ;
     entry PISZE(x: in Integer);
  end;

  task body  PORTIER is
     ilu_czyta: Integer :=  0;  — liczba czytających
  begin
     loop
       select
         accept ZACZYNAM;
         ilu_czyta  :=  ilu_czyta +  1;
       or
         accept KOŃCZĘ;
         ilu_czyta  :=  ilu_czyta - 1;
       or
         when ilu_czyta  =  0 =>
           accept PISZE(x:  in Integer) do
             książka  :=  x;
           end PISZĘ;
```

```
      end select;
    end loop;
  end  PORTIER;

  procedure czytanie(x: out Integer) is
    PORTIER.ZACZYNAM;
    x  := książka;
    PORTIER.KOŃCZĘ;
  end  czytanie;

  procedure pisanie(x:  in  Integer) is
    PORTIER.PISZE(x);
  end  pisanie;
end czytelnia;
```

Pisanie jest możliwe tylko wtedy, gdy zmienna ilu_czyta ma wartość 0. Czytelnicy mogą tu zagłodzić pisarzy, jeśli na tyle często będą wywoływać wejście ZACZYNAM, że nie dopuszczą, by wartość tej zmiennej zmalała do zera. Spotkanie w instrukcji accept PISZĘ odbywa się zawsze tylko z jednym procesem piszącym, pozostałe muszą czekać, aż proces PORTIER znów dojdzie do wykonywania tej instrukcji.

Jeżeli program główny poprzedzimy klauzulą with czytelnia;, to równolegle z tym programem będzie wykonywać się proces PORTIER.

Rozwiązanie poprawne

Zmienimy część implementacyjną procesu PORTIER tak, aby czytanie było możliwe tylko wtedy, gdy nikt nie chce pisać. Można to sprawdzić stosując atrybut count wejścia PISZĘ. Stąd dodatkowy warunek przed spotkaniem w wejściu ZACZYNAM. Unika się w ten sposób zagłodzenia pisarzy. Po zakończeniu pisania natomiast wpuszcza się wszystkich czekających czytelników. Unika się w ten sposób zagłodzenia czytelników.

```
task body PORTIER is
  ilu_czyta: Integer  := 0;  --  liczba czytających
begin
  loop
     select
      when PISZE'count = 0 =>
         accept ZACZYNAM;
         ilu_czyta  :=  ilu_czyta +  1;
     or
      accept KOŃCZĘ;
      ilu_czyta  :=  ilu_czyta - 1;
     or
      when ilu_czyta  = 0 =>
         accept PISZE(x: in Integer)  do
           książka  :=  x
         end PISZĘ;
         loop
           select
             accept  ZACZYNAM;
             ilu_czyta := ilu_czyta + 1;
           else exit;
           end select;
         end loop;
     end select;
  end loop;
end  PORTIER;
```

Zauważmy, że gdy ostatni z czekających czytelników wejdzie do czytelni, proces PORTIER kończy wykonywanie pętli wewnętrznej i przechodzi do wykonywania pętli głównej. Jeśli nadal są czekający pisarze, to wejście do czytelni nowych czytelników jest zablokowane. Jednak wszyscy wpuszczeni wcześniej czytelnicy opuszczą w końcu czytelnię i będzie mógł do niej wejść kolejny pisarz.

6.2.4 Pięciu filozofów

Rozwiązanie z możliwością zagłodzenia

W tym rozwiązaniu synchronizacją procesów FILOZOF zajmuje się proces STÓŁ, który przydziela filozofom jednocześnie oba widelce. W treści procesu FILOZOF użyto atrybutu index, którego wartością jest numer wykonującego tę treść filozofa. Tablica myślący w procesie STÓL służy do zapamiętania, którzy filozofowie aktualnie myślą.

```
task FILOZOF(1. .5)  is
end;
task body FILOZOF is
begin
  loop
    myśli;
    STÓŁ.BIERZE(FILOZOF'index);
    je;
    STÓŁ.ODDAJE(FILOZOF'index);
  end loop;
end FILOZOF;

task STÓŁ is
  entry  BIERZE(1..5);
  entry  ODDAJE(1..5);
end;
task body STÓŁ is
  myślący:  array(1..5) of  Boolean := (others =>  true);
begin
  loop
    for  i in  1..5 loop
      select
         when  myslacy(i)  and  myslacy((i+l) mod  5  + 1) =>
           accept BIERZE(i  mod 5  + 1);
           myslacy(i mod  5  +  1)  := false;
      or
         accept ODDAJE(i);
         myslacy(i) := true;
      else
         null;
      end select;
    end  loop;
  end loop;
end STÓŁ;
```

W procesie STÓŁ zastosowano tablicę pięciu wejść BIERZE i pięciu wejść ODDAJE, po oddzielnej parze dla każdego filozofa. Jest to sposób przekazania procesowi obsługującemu informacji o tym, który z procesów potrzebuje jego usługi. Dzięki temu proces obsługujący może zdecydować zawczasu, który proces obsłużyć. Zauważmy, że gdyby proces przekazywał swój identyfikator jako parametr wejścia, proces obsługujący mógłby go poznać

dopiero w trakcie spotkania, a byloby to już za późno na podejmowanie decyzji o tym, czy można w danej chwili spotykać się z tym procesem, czy też nie.

Rozwiązanie to zaproponowane w artykule [WeLi81], oprócz tego, że nie wyklucza zagłodzenia, ma jeszcze tę wadę, iż wymaga aktywnego czekania od procesu STÓL. Musi on ciągle sprawdzać, czy jakiś filozof może jeść (stąd klauzula else null). Aby uniknąć aktywnego czekania trzeba by pętlę for zastąpić instrukcją

```
select
  when myślący(l) and myslacy(3)   =>
     accept  BIERZE(2);
    myslacy(2) := false;
or
  when myslacy(2) and myslacy(4)   =>
     accept  BIERZE(3);
    myslacy(3) := false;
or
  when myslacy(3) and myslacy(5)   =>
     accept  BIERZE(4);
    myslacy(4) := false;
or
  when myslacy(4) and myślący(l)   =>
     accept  BIERZE(5);
    myslacy(5) := false;
or
  when myslacy(5) and myslacy(2)   =>
     accept  BIERZE(1);
    myślący(l) := false;
or
  accept  ODDAJE(1); myślący(l)  := true;
or
  accept  ODDAJE(2); myslacy(2)  := true;
or
  accept  ODDAJE(3); myslacy(3)  := true;
or
  accept  ODDAJE(4); myslacy(4)  := true;
or
  accept  ODDAJE(5); myslacy(5)  := true;
end  select;
```

co, jak widać, nie jest programowaniem zbyt eleganckim, a ponadto trudnym do realizacji, gdyby filozofów bylo więcej. Mechanizm tablicowania dozorów w CSP zastąpiono w Adzie mechanizmem tablicowania wejść, nie ma już jednak w tym języku możliwości jednoczesnego wstrzymania procesu w oczekiwaniu na spotkanie w którejkolwiek z odpowiednich instrukcji accept.

Rozwiązanie poprawne

Następujące rozwiązanie jest zapisem w języku Ada rozwiązania z rozdz. 3. Każdy proces WIDELEC pelni rolę semafora binarnego (por. p. 6.2.1). Proces LOKAJ jest realizacją w Adzie semafora ogólnego z wartością początkową 4.

```
task LOKAJ is
  entry WEJŚCIE;
  entry WYJŚCIE;
end;
task body LOKAJ  is
  wśrodku: Integer  :=  4;
```

```
begin
  loop
    select
      when wśrodku  > 0 =>
        accept WEJŚCIE;
        wśrodku  :=  wśrodku - 1;
    or
      accept WYJŚCIE;
      wśrodku  :=  wśrodku + 1;
    end select;
  end loop;
end LOKAJ;

task WIDELEC(1..5)  is
  entry BIERZE;
  entry ODDAJE;
end;
task body WIDELEC is
begin
  loop
    accept BIERZE;
    accept ODDAJE;
  end loop;
end WIDELEC;

task FILOZOF(1..5)  is
end;
task body FILOZOF is
begin
  loop
    myśli;
     LOKAJ.WEJŚCIE;
     WIDELEC(FILOZOF'index).BIERZE;
     WIDELEC(FILOZOF'index mod  5+l).BIERZE;
     je;
     WIDELEC(FILOZOF'index).ODDAJE;
     WIDELEC(FILOZOF'index mod  5+l).ODDAJE;
     LOKAJ.WYJŚCIE;
   end loop;
 end;
```

6.3 Zadania

Proponujemy do rozwiązania zadania poświęcone realizacji w Adzie mechanizmu semaforów i monitorów oraz dwa wybrane zadania z rozdz. 4 i trzy zadania z rozdz. 5. Czytelnika pragnącego lepiej poznać możliwości języka Ada zachęcamy do zapisania w nim także rozwiązań innych zadań. Przed rozwiązaniem zadania 6.3.2 należy zapoznać się z definicją semafora w systemie Unix podaną w rozdz. 8.

6.3.1 Implementacja semafora dwustronnie ograniczonego

Zapisz w Adzie proces, który jest realizacją semafora dwustronnie ograniczonego. Przyjmij, że podczas inicjacji semafora ustala się zarówno jego wartość początkową, jak i ograniczenie górne.

6.3.2 Implementacja semafora unixowego

Zapisz w Adzie proces SEM_UNIX, który będzie symulował semafor unixowy S (por. rozdz. 8). Proces ten powinien realizować operacje semaforowe P(S), V(S,m), Z(S), nP(S,m), nZ(S) i funkcje semaforowe wart(S), czekP(S), czekZ(S) oraz umożliwiać nadawanie wartości semaforowi w dowolnej chwili. Zastanów się nad możliwością realizacji operacji P(S,m).

6.3.3 Implementacja monitora ograniczonego

Napisz proces CONDITION realizujący monitorowy obiekt typu condition i udostępniający operacje wait, signal i empty, przy założeniu, że signal jest zawsze ostatnią wykonywaną operacją procedury monitorowej.

6.3.4 Zasoby dwóch typów

Zapisz w Adzie proces MON, który jest rozwiązaniem zadania 4.3.10. Powinien on udostępniać wejściaPRZYDZIELi, i=l,2,3, realizujące przydział zasobu procesowi odpowiedniej grupy oraz wejście ZWALNIAM realizujące zwolnienie zasobu podanego typu.

6.3.5 Szeregowanie żądań do dysku

Zapisz w Adzie proces DYSK, który jest rozwiązaniem zadania 4.3.13. Powinien on udostępniać dwa wejścia PRZYDZIEL i ZWOLNIJ realizujące odpowiednio przydział i zwolnienie dostępu do dysku. Uwzględnij obie wersje, tzn. szeregowanie według strategii FCFS i SCAN.

6.3.6 Algorytm Ricarta i Agrawali

Zapisz w Adzie algorytm Ricarta i Agrawali (opisany w p. 5.2.1) uwzględniając jednocześnie korygowanie logicznych zegarów (zad. 5.3.5). Załóż, że procesy POMOCNIK komunikują się ze sobą przez systemowy proces SIEĆ realizujący fizyczne przesłania komunikatów między różnymi komputerami (komunikaty te mogą być przesyłane asynchronicznie, a więc bez udziału mechanizmu spotkań). SIEĆ udostępnia dwa wejścia: ZADANIE(t: in integer; j: in integer), którego wywołanie powoduje wysłanie komunikatu postaci (t,i,j) do procesu j (t jest lokalnym logicznym czasem, a i numerem procesu-nadawcy), oraz ODPOWIEDZ(j : out integer), którego wywołanie powoduje wysłanie komunikatu (i , j) do procesu j (i jest numerem procesu-nadawcy). Po otrzymaniu komunikatu postaci (t,i,j) proces SIEĆ wywołuje wejście ODBIERZ_ZADANIE(t, i) w procesie POMOCNIK(j), a po otrzymaniu komunikatu postaci (i,j) wywołuje w nim wejście ODBIERZ_ODPOWIEDZ.

6.3.7 Centrala telefoniczna

Zapisz w Adzie procesy NADAWCA, ODBIORCA i CENTRALA, które są rozwiązaniem zadania 5.3.8. ODBIORCA powinien udostępniać wejście TELEFON realizujące przyjęcie

rozmowy przez nadawcę. Proces CENTRALA powinien udostępniać wejścia DZWONIĘ i ZWALNIAM realizujące reakcję centrali na żądanie nadawcy odpowiednio nawiązania połączenia i zerwania połączenia.

6.3.8 Sito Eratostenesa

Zapisz w Adzie proces SITO, który jest rozwiązaniem zadania 5.3.14. Proces ten powinien udostępniać wejście LICZBA, przez które przekazuje mu się liczbę. (Proces SITO(0) zapisz oddzielnie.) Załóż istnienie procesu:

```
task  WYJŚCIE is
   entry PISZ(x:  in Integer);
end;
```

który liczby otrzymywane w wejściu PISZ wypisuje na urządzenie wyjściowe (np. ekran).

6.4 Rozwiązania

6.4.1 Implementacja semafora dwustronnie ograniczonego

Proces SEM_2_OGR udostępnia oprócz wejść PD i VD także wejście INICJACJA służące do nadania semaforowi wartości początkowej i ustalenia ograniczenia górnego. Pomijamy tu kontrolę poprawności wywołania tego wejścia (nie sprawdzamy, czy 0 < x < k).

```
task  SEM_2_OGR is
  entry  INICJACJA(x,k:  in  Integer);
  entry  PD;
  entry  VD;
end;
task  body SEM_2_OGR is
  n:  Integer;   -- górne ograniczenie
  s:  Integer;   -- wartość semafora
begin
  accept INICJACJA(x,k:  in Integer) do
    s := x;
    n := k;
  end INICJACJA;
  loop
    select
       when s  > 0  => accept PD; s  := s -  1;
    or
       when s  < n  => accept VD; s  := s +  1;
    end  select;
  end loop;
end SEM_2_OGR;
```

Proces SEM_2_OGR ma strukturę podobną do procesu BUFOR z przykładu 6.2.2. Operacja VD odpowiada wstawianiu do bufora, operacja PD — pobieraniu z bufora, a wartość s wskazuje liczbę wolnych miejsc w buforze.

6.4.2 Implementacja semafora unixowego

Ponieważ operację nadania wartości semaforowi unixowemu można wykonać w każdej chwili, należy ją akceptować zarówno na początku wykonywania procesu SEM_UNIX, jak i wewnątrz głównej pętli. Operacja Z ma tę własność, że w chwili, gdy semafor osiągnie wartość 0, wszystkie procesy wstrzymane na operacji Z kończą jej wykonywanie. Stąd w procesie SEM_UNIX po spotkaniu się z jednym procesem w wejściu Z trzeba spotkać się natychmiast ze wszystkimi oczekującymi tam procesami.

```
task SEM_UNIX is
  entry NADANIE_WART(x: in Integer);
  entry P;
  entry V(m: in Integer);
  entry Z;
  entry nP(m: in Integer; b:  out Boolean);
  entry nZ(b: out Boolean);
  entry CZEKP(w: out Integer);
  entry CZEKZ(w: out Integer);
  entry WART(w: out Integer);
end;

task body SEM_UNIX is
  s: Integer;
begin
  accept NADANIE_WART(x: in Integer)  do
    s := x;
  end NADANIE_WART;
  loop
    select
      when s > 0 => accept P; s := s  - 1;
    or
      accept V(m: in Integer); do s  := s + m; end V;
    or
      when s = 0 => accept Z;
        while Z'count > 0 loop
          accept Z;
        end loop;
    or
      accept nP(m: in Integer; b: out Boolean)  do
        b := m <= s;
        if b then s := s - m; end if;
      end nP;
    or
      accept nZ(b: out Boolean) do b  := s = 0;  end nZ;
    or
      accept CZEKP(w:  out Integer) do
        w := P'count;
      end CZEKP;
     or
       accept  CZEKZ(w:  out Integer)  do
         w  : =  Z'count;
       end  CZEKZ;
     or
       accept  WART(w: out  Integer) do w := s;  end WART;
     or
       accept  NADANIE_WART(x: in Integer) do
         s  :=  x;
       end  NADANIE_WART;
     end select;
```

```
    end loop;
  end SEM_UNIX;
```

Dostępna w systemie Unix operacja P(S,m), która umożliwia wstrzymanie procesu w oczekiwaniu na dowolnie wybraną wartość m semafora S, nie daje się zrealizować w Adzie w prosty sposób. Problem polega na tym, że aby dowiedzieć się, na jaką. wartość czeka proces, trzeba najpierw spotkać się z nim. Jeśli w trakcie spotkania okaże się, że semafor nie ma odpowiedniej wartości, spotkanie to musi się zakończyć, aby proces SEM_UNIX mógł zaakceptować wejście V. Jednak zakończenie spotkania oznacza zakończenie wstrzymywania procesu obsługiwanego. W przykładzie 6.2.4 mieliśmy zbliżony problem i użyliśmy tam tablicy wejść. Tutaj można zrobić podobnie.

Jeśli zakres wartości parametru m jest bardzo ograniczony lub z góry wiadomo, jakie to będą wartości, to odpowiednie instrukcje acceptP(m: in Integer)można jawnie wypisac. Na przykład implementując w Adzie semafor M z przykładu 8.2.3, wystarczy dopisać w instrukcji:

```
    select gałąź
  when s>=N  => accept P(m: in Integer) do s := s - N; end P;
```

ponieważ wiadomo, że wartością m będzie zawsze N. Jeśli zakres jest większy, trzeba skorzystać z pętli for takiej jak w przykładzie 6.2.4, wymusza to jednak aktywne czekanie.

Jeśli wszystkie procesy wykonujące operację P(S, m) na danym semaforze są umieszczone w jednej tablicy o indeksach 1. .N, to można rozbić tę operację na dwa etapy. W pierwszym proces wywołujący przekazuje procesowi SEM_UNIX wartość m oraz swój indeks i, a w odpowiedzi otrzymuje informację, czy operacja P dala się wykonać natychmiast. Jeśli nie, to proces przechodzi do drugiego etapu, w którym wywołuje wejście DALEJ(i). Część specyfikacyjną procesu SEM_UNIX należy rozszerzyć o następujące wejścia:

```
  entry P(m:  in  integer;  i:  in integer,
          przeszło:  out  boolean);
  entry DALEJ(1..N);
```

W instrukcji select należy dodać gałąź

```
 accept P(m: in integer; i:  in  integer;
           przeszło:  out boolean)  do
   przeszło := mozna_przepuscic(m,i);
 end P;
```

oraz zmodyfikować implementację wejścia V tak, aby wznawiać tyle procesów, ile w danej chwili można wznowić:

```
 accept V(m: in integer) do
   s  := s + m;
   while mozna_wznowic(i)  loop
     accept DALEJ(i);
   end  loop;
 end V;
```

Funkcja logiczna mozna_przepuscic zmniejsza s o m, gdy jest to możliwe i wówczas zwraca wartość true, w przeciwnym razie zwraca wartość false zapamiętując w pomocniczych strukturach indeks procesu i oraz jego zapotrzebowanie m. Wartością funkcji logicznej mozna_wznowic jest odpowiedź na pytanie, czy można wznowić (zgodnie z przyjętą strategią) jakiś proces. Jeśli tak, to zmniejsza ona s o zapotrzebowanie tego procesu i zwraca jego indeks i. Trzeba również usunąć wejście P i odpowiednio zmodyfikować implementację wejścia CZEKP.

6.4.3 Implementacja monitora ograniczonego

Rolę kolejki procesów czekających na warunek spełnia kolejka procesów czekających w wejściu WAIT. W związku z tym, jeżeli żaden proces nie czeka na spotkanie w wejściu WAIT, to można zaakceptować spotkanie w wejściu SIGNAL bez dalszych konsekwencji. Będzie to sygnał wysłany „w powietrze". Jeśli natomiast jakiś proces czeka na spotkanie w WAIT, to również można zaakceptować spotkanie w wejściu SIGNAL, ale podczas tego spotkania proces CONDITION musi zrealizować także spotkanie w wejściu WAIT. Atrybut count wejścia WAIT służy do stwierdzenia, ile procesów oczekuje na spełnienie warunku.

```
task CONDITION is
  entry WAIT;
  entry SIGNAL;
  entry EMPTY(b:  out  Boolean);
end;
task body CONDITION  is
begin
  loop
    select
      when WAIT'count =  0  =>
        accept SIGNAL;
    or
      when WAIT'count >  0  =>
        accept SIGNAL do
          accept WAIT;
        end SIGNAL;
    or
      accept EMPTY(b: out  Boolean)  do
        b := WAIT'count  =  0;
      end EMPTY;
    end select;
  end loop;
end CONDITION;
```

6.4.4 Zasoby dwóch typów

Wejście PRZYDZIELljest bezparametrowe, bo zawsze jest przydzielany zasób typu A. Wejście PRZYDZIEL2 ma jeden parametr określający typ przydzielonego zasobu. Wejście PRZYDZIEL3 ma dwa parametry wyjściowe. Pierwszy określa, czy zasób przydzielono, drugi podaje typ przydzielonego zasobu.

```
M: constant Integer  :=  ?;   —  liczba zasobów typu A
N: constant Integer  :=  ?;   --  liczba zasobów typu B

task MON is
  entry PRZYDZIELI;
  entry PRZYDZIEL2(typ:  out A..B);
  entry PRZYDZIEL3(typ:  out A..B;  dostał: out Boolean);
  entry ZWALNIAM(typ: in A..B);
end;

task body MON is
  wolneA: Integer  := M;
  wolneB: Integer  := N;
begin
  loop
```

```
      select
        when wolneA > 0 =>
           accept PRZYDZIELI;
           wolneA := wolneA - 1;
      or
        when wolneA + wolneB > 0 =>
           accept PRZYDZIEL2(typ: out  A..B)  do
             if  wolne A > 0 then
              typ := A;
              wolneA := wolneA - 1;
             else
              typ := B;
              wolneB := wolneB - 1;
             end if;
           end PRZYDZIEL2;
      or
        accept  PRZYDZIEL3(typ: out A..B;  dostał: out Boolean)
        do
           if wolneB > 0 then
             typ := B;
             wolneB := wolneB - 1;
             dostal := True;
           elsif wolneA > 0 then
             typ := A;
             wolneA := wolneA - 1;
             dostal := True;
           else
             dostal := False;
           end if;
        end  PRZYDZIEL3;
      or
        accept  ZWALNIAM(typ: in A..B)  do
           if typ = A then wolneA := wolneA  + 1;
           else  wolneB := wolneB + 1;
           end if;
        end  ZWALNIAM;
      end  select;
    end loop;
  end MON;
```

Rozwiązanie to różni się od rozwiązania z rozdz. 4 tym, że procesy żądające zasobu nie czekają w jednej lecz w trzech kolejkach. W rezultacie wybór obsługiwanej grupy jest niedeterministyczny. Procesy grupy trzeciej mogą być zawsze obsłużone, procesy grupy drugiej tylko wtedy, gdy są jakieś wolne zasoby, a procesy grupy pierwszej tylko wtedy, gdy są wolne zasoby typu A. Procesy grupy pierwszej nie mają tu tak zdecydowanego priorytetu jak w rozwiązaniu z p. 4.4.10. W przypadku zwalniania zasobu typu A nie przydziela się go bezwarunkowo czekającemu procesowi grupy pierwszej; jeśli czekają także procesy grupy drugiej, to wybórjest niedeterministyczny, jest bowiem spełniony zarówno warunek wolneA > 0, jak i warunek wolneA + wolneB > 0. Co więcej, jeśli w chwili zwracania zasobu typu A pojawi się żądanie procesu grupy trzeciej, to również ono może być obsłużone. Mimo tych wszystkich różnic przedstawione rozwiązanie jest poprawnym rozwiązaniem postawionego zadania.

6.4.5 Szeregowanie żądań do dysku

Rozwiązanie wersji 1

Realizacja pierwszej wersji zadania jest banalna, ponieważ w Adzie procesy czekają na spotkanie w kolejce obsługiwanej zgodnie z dyscypliną FCFS. (W istocie jest to realizacja semafora binarnego.)

```
task DYSK is
   entry PRZYDZIEL;
   entry ZWOLNIJ;
end;
task body DYSK  is
begin
   loop
     accept PRZYDZIEL;
     accept ZWOLNIJ;
   end  loop;
end DYSK;
```

Rozwiązanie wersji 2

W Adzie nie ma możliwości priorytetowego szeregowania procesów w oczekiwaniu na spotkanie. Musimy więc zastosować tablicę wejść umożliwiająca określanie z góry numeru cylindra, do którego żądanie można zaakceptować. Takie podejście prowadzi jednak do aktywnego czekania w procesie DYSK w wtedy, gdy nie ma żadnych żądań do dysku.

```
C: constant Integer := ?; —  liczba cylindrów

task DYSK is
   entry PRZYDZIEL(1..C);
   entry ZWOLNIJ;
end;
task  body DYSK is
begin
  loop
    for i in l..C  loop  — od krawędzi do środka cylindra
       while PRZYDZIEL(i)'count  >  0 loop -- obsługa kolejki
         accept PRZYDZIEL(i);
         accept ZWOLNIJ;
       end loop;
    end loop;
    for i in reverse l..C loop  — od środka  do  krawędzi
       while PRZYDZIEL(i)'count  >  0 loop
         accept PRZYDZIEL(i);
         accept ZWOLNIJ;
       end loop;
    end loop;
  end loop;
end DYSK;
```

Ponieważ pojedyncze wykonanie instrukcji accept obsługuje tylko jedno spotkanie, musimy w pętli for umieścić instnikcję while, która umożliwi obsługę wszystkich procesów oczekujących na transmisję z tego samego cylindra. Podobnie jak w pierwszym rozwiązaniu tego zadania w rozdz. 4, nieprzerwany strumień żądań do jednego cylindra może spowodować zagłodzenie innych procesów. Łatwo tę wadę usunąć. Wystarczy przed rozpoczęciem obsługi żądań do danego cylindra zapamiętać ich liczbę, up. w zmiennej j, a pętlę while zamienić na pętlę for, która będzie wykonana dokładnie j razy.

6.4.6 Algorytm Ricarta i Agrawali

Prezentowane rozwiązanie jest modyfikacją rozwiązania przedstawionego w książce [Ma0087]. Różnica polega na zastosowano zagnieżdżonych instrukcji accept. Proces, który chce wejść do sekcji krytycznej, jest w trakcie spotkania ze swoim procesem POMOCNIK tak długo, aż ten nie odbierze potwierdzeń od wszystkich pozostałych pomocników. W tym czasie POMOCNIK może także odbierać od procesu SIEĆ żądania wejścia do sekcji krytycznej innych procesów. Odpowiada na nie w zależności od czasu i numeru nadawcy tych żądań. Żądania te odbiera także poza instrukcją accept CHCĘ, ale wówczas zawsze odpowiada na nie natychmiast (proces P nie chce wówczas wejść do swojej sekcji krytycznej). Zauważmy również, że odpowiedzi z sieci mogą przychodzić tylko wtedy, gdy proces jest w trakcie wykonywania instrukcji accept CHCĘ, zwolnienie natomiast jest możliwe tylko poza tą instrukcją. (Uwaga: relację „różne" w Adzie zapisuje się znakiem /=.) N: constant Integer := ?; — liczba procesów

```
task P(1..N) is
end;

task body P is
begin
  loop
    wlasne_sprawy(P'index);
    POMOCNIK(P'index).CHCĘ;
    sekcja_krytyczna(P'index);
    POMOCNIK(P'index).ZWALNIAM;
  end loop;
end P;

task POMOCNIK(1..N) is
  entry CHCĘ;
  entry ZWALNIAM;
  entry'ODBIERZ_ZADANIE(czas:   in Integer;
                        odkogo:  in Integer);
  entry ODBIERZ_ODPOWIEDZ;
end;

task body POMOCNIK is
  mojt: Integer := 0;          — lokalny logiczny czas;
  t: Integer := 0;            — największy  czas
  licz: Integer := 0;          — liczba odpowiedzi
  wstrzymany:  array(l..N)  of Boolean := (others => False);
                              — kt6re procesy
                              — wstrzymał ten proces
begin
  loop
    select
      accept CHCĘ do
        mojt := t + 1;
        licz := 0;
        for j  in l..N  loop    — wysłanie żądań do
                              — pozostałych procesów
          if j /= POMOCNIK'index then
            SIEĆ.ŻĄDANIE(mójt.j);
          end if;
        end loop;
        loop
          select
            accept ODBIERZ_ZADANIE(czas:    in Integer;
```

```
                                odkogo:  in Integer)  do
            if czas > t then t := czas
            end if;          -- skorygowanie czasu
            if (czas > mójt) or ((czas  =  mójt)  and
               (odkogo > POMOCNIK'index))
            then             — niższy priorytet
              wstrzymany(odkogo)  := True;
            else             — odpowiedź  pozytywna
              SIEĆ.ODPOWIEDŹ(odkogo);
            end if;
          end ODBIERZ_ZADANIE;
        or
          accept ODBIERZ_ODPOWIEDZ;
          exit when licz = N - 2;
                             — już wszystkie odpowiedzi
          licz := licz + 1;
        end select;
      end loop;
    end CHCĘ;
                             — nowe żądania zaczekają w
                             — kolejce  do ODBIERZ_ZADANIE
    accept ZWALNIAM;
    for j in 1..N loop
      if wstrzymany(j) then -- odpowiedzi do wszystkich
        SIEĆ.ODPOWIEDŹ(j);  — wstrzymanych
        wstrzymany(j) := False;
      end if;
    end loop;
  or
    accept ODBIERZ_ZADANIE(czas:  in Integer;
                           odkogo: in Integer)  do
      if czas > t then t := czas
      end if;                —. skorygowanie czasu
      SIEĆ.ODPOWIEDŹ(odkogo);
                             — odpowiedź  pozytywna
    end ODBIERZ_ZADANIE;
  end select;
 end loop;
end POMOCNIK;
```

6.4.7 Centrala telefoniczna

Oto zapisane w Adzie rozwiązanie z p. 5.4.8.

```
N:  constant Integer := ?;           — liczba nadawców
M:  constant Integer := ?;           — liczba odbiorców

task NADAWCA(1..N) is
end;

task body NADAWCA is
  m,k: Integer;
begin
  loop
    losuj(m);                        -- wybór rozmówcy
    loop
      CENTRALA.DZWONIE(m,k);
      exit when k /= 0;              — wykręcanie numeru
    end loop;
```

```
    nadaj(k);                          — rozmowa
    CENTRALA.ZWALNIAM(k);              — odłożenie słuchawki
  end loop;
end NADAWCA;

task ODBIORCA(l..M) is
  entry TELEFON(x: in Integer);
end;

task body ODBIORCA is
  k: Integer;
begin
  loop
    accept TELEFON(x: in Integer) do
      k := x;                          — podniesienie
    end TELEFON;                        — słuchawki
    odbierz(k);                        — rozmowa
  end loop;
end ODBIORCA;

task CENTRALA is
-  entry DZWONIE(m: in Integer; k: out Integer);
  entry ZWALNIAM(k: in Integer);
end;
task body CENTRALA  is
  K: constant  Integer  = ?;            — liczba łączy
  m,i: Integer;
  wolne: Integer  := K;                 — 1. wolnych łączy
  kanał: array(l..K) of Integer := (others => 0);
                                        — kto odbiera przez
                                        — dane łącze
  odbiór: array(l..M)  of boolean := (others => False);
                                        — czy odbiorca mówi
begin
  loop
    select
      accept DZWONIE(m: in  Integer; k: out Integer) do
        if wolne  =  0 or else odbiór(m) then k := 0
        else
          i  := 1;
          wolne  :=  wolne -  1;        — zabranie łącza
          loop
            if kanał(i) /=  0 then i := i + 1
            else  exit;                 — szukanie wolnego
            end if;                     — łącza
          end  loop;
          odbiór(m)  := True;           — ODBIORCA(m) zajęty
          kanał(i)  :=  m;              — łącze i-te zajęte
          ODBIORCA(m).TELEFON(i);       — dzwoni u odbiorcy
          k  := i;
        end if;
      end DZWONIĘ;
    or
      accept ZWALNIAM(k: in Integer) do
        odbior(kanal(k)) :  = False;    — odbiorca wolny
        kanał(k)  := 0;                 — łącze jest wolne
        wolne  :=  wolne + 1;           — i wraca do puli
      end ZWALNIAM;
    end select;
  end loop;
end CENTRALA;
```

Zauważmy, że w rozwiązaniu w Adzie stwierdzenie faktu, czy uzyskano połączenie, odbywa się podczas jednego spotkania w wejściu DZWONIĘ. W czasie tego spotkania nadawca przekazuje centrali numer odbiorcy, z którym chce rozmawiać, centrala sprawdza, czy ma wolne łącze i czy odbiorca jest wolny. Jeśli tak, to przekazuje odbiorcy numer łącza, po czym, w zależności od sytuacji, zwraca nadawcy zero hib numer łącza. W Adzie nie mamy możliwości niedeterministycznego wyboru łącza, tak jak w CSP.

6.4.8 Sito Eratostenesa

W przeciwieństwie do rozwiązania w CSP z p. 5.4.14 proces generujący liczby nie musi nazywać się tak samo jak pozostałe procesy, proces SITO bowiem i tak nie wie, od jakiego procesu otrzymuje liczby.

```
M: constant  Integer := ?;            — liczba sit

task GENERATOR  is
end;

task body GENERATOR is
  N: constant Integer := ?;          — zakres liczb
  i: Integer;
begin
  WYJSCIE.PISZ(2);
  i := 3;
  while i <= N  loop
    SITO(l).LICZBA(i);
    i  := i + 2;
  end loop;
end GENERATOR;

task SITO(1..M)  is
  entry LICZBA(x:  in Integer);
end;

task body SITO  is
  i,k: Integer;
begin
  accept LICZBA(x:  in Integer)  do
    i  := x;
  end LICZBA;
  WYJŚCIE.PISZ(i);
  loop
    select
      accept LICZBA(x:  in Integer) do
        k := x;
      end LICZBA;
      if k mod  i /= 0 then
         if SITO'index = M then WYJŚCIE.PISZ(k);
         else SITO(SITO'index+l).LICZBA(k);
         end if;
       end if;
     else
       terminate;
     end select;
  end  loop;
end  SITO;
```

Ponieważ, zgodnie z umową, SITO'index jest indeksem procesii SITO, w którym wykonują się instrukcje, więc wyrażenie SITO(SITO'index+l) wskazuje kolejny proces SITO w tablicy procesów. Dzięki zastosowaniu klauzuli else terminate proces SITO(1) zakończy się, gdy zakończy się proces GENERATOR, po czym zakończą się pozostałe procesy SITO.

W podobny sposób jak sito Eratostenesa można zapisać w języku Ada rozwiązania zadań 5.3.9, 5.3.11, 5.3.15 i 5.3.16, które także mają strukturę potoku.

7 Przestrzeń krotek w Lindzie

7.1 Wprowadzenie

7.1.1 Przestrzeń krotek

W roku 1985 D. Gelernter opublikował artykuł [Gele85], w którym zaproponował nowy mechanizm komunikacji między procesami nazwany wdzięcznym imieniem Linda. W odróżnieniu od synchronicznych spotkań w CSP lub Adzie, które z jednej strony wymagają, aby komunikujące się procesy istniały jednocześnie w czasie, a z drugiej zakładają, że procesy te znają swoje nazwy (w obie strony w CSP, w jedną stronę w Adzie), komunikacja w Lindzie nie narzuca żadnych powiązań ani w czasie, ani w przestrzeni adresowej.

Linda nie jest jakimś konkretnym językiem programowania. Jest to jedynie pewna idea, której realizację można włączyć do dowolnego języka programowania współbieżnego. Przykłady i rozwiązania zadań będziemy zapisywać tutaj w notacji języka PascaLC.

Podstawowym pojęciem Lindy jest krotka (tuple)1. Krotka to ciąg danych (o określonej lub nieokreślonej wartości, por. 7.1.4), z których każda ma określony typ. Ciąg typów poszczególnych elementów krotki tworzy jej sygnaturę. Na przykład krotka

```
(5, 3.14, true, 'c', 20)
```

ma sygnaturę

```
(integer, real, boolean, char, integer)
```

Typami elementów krotek mogą być także typy złożone, jak napisy, tablice, rekordy.

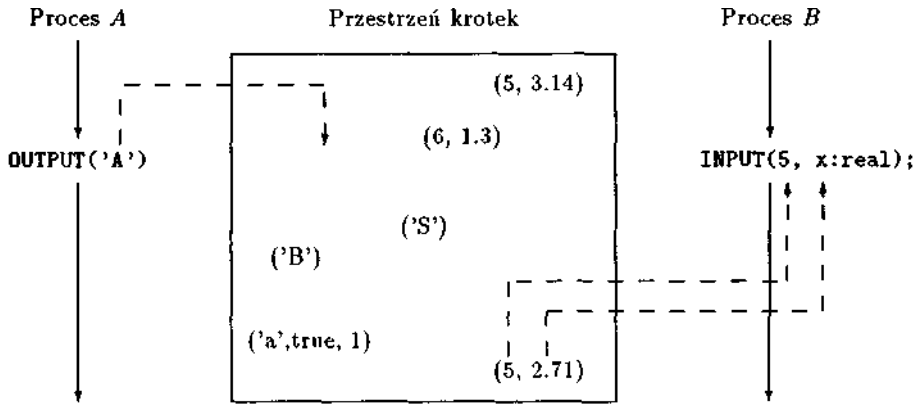
W Lindzie wszystkie procesy realizują się w środowisku zwanym przestrzenią krotek. Z przestrzeni tej proces pobiera dane w formie krotek i do tej przestrzeni wysyła wyniki, także w formie krotek. Ponieważ każdy proces komunikuje się jedynie z otaczającą go przestrzenią, więc nie musi nic wiedzieć o istnieniu innych procesów (nie musi w ogóle znać ich nazw). Do komunikacji z przestrzenią krotek służą operacje: INPUT, OUTPUT, READ, TRY_INPUT, TRY_READ.

7.1.2 Operacja INPUT

Pobieranie krotek z przestrzeni realizuje się za pomocą operacji INPUT. Jej parametrami są nazwy zmiennych z podanymi typami (w p. 7.1.4 podamy jeszcze inne możliwości). W wyniku wykonania tej operacji z przestrzeni jest pobierana krotka o sygnaturze zgodnej z sygnaturą ciągu parametrów tej operacji. Tak więc do pobrania krotki (5, 3.14, true, 'c', 20) można użyć następującej operacji:

```
INPUT(i:integer, r:real, b:boolean, c:char, j:integer).
```

Po jej wykonaniu zmienne występujące jako parametry operacji INPUT będą miały następujące wartości: i=5, r=3.14, b=true, c='c', j=20.
Jeśli w chwili wywołania operacji INPUT w przestrzeni nie ma krotki o wskazanej sygnaturze, proces wywołujący jest wstrzymywany do czasu, w którym taka krotka się pojawi. Powstaje oczywiście problem, co się dzieje, gdy w przestrzeni pojawi się krotka, na którą czeka więcej niż jeden proces. Podobnie jak w przypadku semaforów, nie będziemy nic zakładać o kolejności wznawiania procesów. Będziemy natomiast wymagać, aby wznawianie było realizowane w sposób nie powodujący zagłodzenia, tzn. że jeśli krotka, na którą czeka proces, pojawia się w przestrzeni dowolną liczbę razy, to proces ten będzie w końcu wznowiony. Inaczej mówiąc, realizacja operacji INPUT musi być żywotna.



Rys. 7.1. Komunikacja przez przestrzeń krotek

7.1.3 Operacja OUTPUT

Krotki wysyla się w przestrzeń za pomocą operacji OUTPUT. Na przykład

```
OUTPUT(5, 3.14, true, 'c', 20)
```

spowoduje umieszczenie w przestrzeni podanej krotki. Operację OUTPUT można wykonać w każdej chwili, nigdy nie blokuje ona procesu. Jeśli na liście parametrów operacji OUTPUT umieścimy wyrażenie, to utworzona krotka będzie miała na odpowiedniej pozycji wartość równą aktualnej wartości podanego wyrażenia. Sposób komunikacji przez przestrzeń krotek za pomocą operacji INPUT i OUTPUT ilustruje rys. 7.1.

7.1.4 Wybór selektywny

Operacje INPUT i OUTPUT zachowują się bardzo podobnie jak zwykle pascalowe operacje read i write. Jednak w rzeczywistości zaproponowany w Lindzie mechanizm komunikacji jest znacznie mocniejszy dzięki dopuszczeniu drobnych odstępstw od podanych wyżej zasad. Po pierwsze, parametrem operacji OUTPUT może być także niezainicjowana zmienna ze wskazanym typem. Wykonanie operacji OUTPUT z takim parametrem powoduje utworzenie krotki z elementem o nieokreślonej wartości, ale ze wskazanym typem. Po drugie, parametrem operacji INPUT może być stała lub zmienna o zadanej wartości. Wykonanie operacji INPUT z takim parametrem spowoduje pobranie krotki, która na odpowiedniej pozycji ma wskazaną wartość. Na przykład w wyniku operacji

```
INPUT(i:integer,  3.14, b:boolean,  c, j:integer)
```

z przestrzeni zostanie pobrana taka krotka, która na pozycji drugiej ma liczbę 3.14, a na czwartej znak równy aktualnej wartości zmiennej c. W ten sposób ze zbioru krotek o tej samej sygnaturze można wyselekcjonować krotki o zadanych wartościach.

„Dziurawą" krotkę z nieokreśloną wartością na pewnej pozycji można pobrać za pomocą operacji INPUT, w której na pozycji „dziury" wskaże się konkretną wartość podanego typu. Na przykład w wyniku wykonania operacji

```
OUTPUT(i:integer, 3.14, true, 'c', 20)
```

powstaje krotka, która może być pobrana w wyniku wykonania operacji

```
INPUT(5, r:real, true, c:char, j:integer).
```

choć tą samą operacją INPUT można pobrać krotkę mającą na pierwszej pozycji wartość 5.

Jeżeli pierwszy element krotki oznacza numer procesu, do którego jest ona skierowana, to krotka utworzona w powyższej operacji OUTPUT może być pobrana przez proces o dowolnym numerze, podczas gdy krotka z określoną liczbą na początku może być przeczytana tylko przez określony proces. Dzięki takiemu mechanizmowi można kierować żądania do wskazanego lub dowolnego procesu.

Jeśli w chwili wykonywania operacji INPUT w przestrzeni znajduje się wiele „pasujących" krotek, to będzie pobrana jedna z nich w sposób niedeterministyczny. (Dotyczy to również przypadku, gdy w przestrzeni znajduje się zarówno krotka, która na wskazanej pozycji ma wartość nieokreśloną, jak i krotka, która na wskazanej pozycji ma wartość podaną w operacji INPUT.) Zakładamy tu również, że nie jest możliwe zagłodzenie krotek, tzn., jeśli dostatecznie dużo razy jest wykonywana operacja INPUT z sygnaturą zgodną z sygnaturą pewnej krotki umieszczonej w przestrzeni, to w końcu ta krotka będzie z przestrzeni pobrana. Zauważmy, że wymaganie to różni się od wymagania sformułowanego w p. 7.1.2 odnośnie żywotności operacji INPUT.

7.1.5 Operacja READ

Oprócz operacji INPUT i OUTPUT można także wykonywać operację READ, która ma taki sam efekt jak INPUT z tą różnicą, że przeczytana krotka nie jest usuwana z przestrzeni, dzięki czemu mogą ją również odczytać inne procesy wykonujące operację READ. W ten sposób można realizować rozgłaszanie informacji.

Istnieją również nieblokujące wersje operacji INPUT i READ, realizowane za pomocą funkcji logicznych TRY_INPUT i TRY_READ, które zwracają wartość true, jeśli pobranie lub wczytanie krotki zakończyło się powodzeniem, a w przeciwnym razie zwracają wartość false. Zainteresowanych bliżej Lindą odsyłamy do artykułów [Gele85, AhCG86, CzZi93] oraz książki [BenA90].

7.1.6 Ograniczenia

Opisana przestrzeń krotek jest czymś w rodzaju wielowymiarowego bufora, do którego wstawia się porcje wykomijąc operację OUTPUT i z którego pobiera się je za pomocą operacji INPUT. Dzięki temu w Lindzie dnżo prościej zapisuje się problemy wymagające komunikacji przez bufor lub przez innego rodzaju pośrednika. Należy jednak podkreślić, że sama przestrzeń nie zapewnia zachowania kolejności przekazywanych krotek. Jeśli kolejność ta jest

istotna, należy każdą krotkę rozszerzyć o jej numer podawany przez proces wykonujący operację OUTPUT. Proces, który wykonuje operację INPUT, musi jawnie wskazać numer pobieranej krotki. Podobnie, jeśli krotka ma być odebrana tylko przez określony proces (lub tylko od określonego procesu), numer identyfikujący ten proces musi być częścią krotki.

Wybór selektywny umożliwia pobranie krotki o zadanej wartości na wskazanej pozycji. Nie ma jednak możliwości podania warunku, jaki powinna spełniać wartość na wskazanej pozycji pobieranej krotki (np. pobierz krotkę, której pierwszy element jest nieujemny). Obowiązek sprawdzania odpowiedniego warunku spoczywa więc na procesie tworzącym krotkę, który powinien rozszerzyć ją o element zawierający odpowiednią informację dla odbiorcy (por. uwaga do rozwiązania 7.4.8).

7.2 Przykłady

7.2.1 Wzajemne wykluczanie

Pokażemy jak w Lindzie można zrealizować semafor ogólny oraz jak realiziije się mechanizm przekazywania uprawnień.

Semafor ogólny

Deklaracji semafora postaci

```
var  sem:   semaphore := N;
```

odpowiada wprowadzenie do przestrzeni krotek przez specjalny proces inicjujący N krotek postaci ('sem'):

```
for  i := 1 to  M  do OUTPUT('sem');
```

Wówczas wykonanie operacji wejścia INPUT('sem') jest równoważne wykonaniu P(sem), a wykonanie operacji wyjścia OUTPUT('sem') jest równoważne wykonaniu V(sem).

Przekazywanie uprawnień

Na początku działania systemu trzeba do przestrzeni wprowadzić krotkę z uprawnieniem wykonując operację

```
OUTPUT('uprawnienie')
```

Zgodnie z zasadą opisaną w p. 5.2.1 uprawnienie powinno być przekazywane od procesu do procesu. Jednak w Lindzie każdy proces komunikuje się jedynie z otaczającą go przestrzenią. Przestrzeń krotek jest więc tu czymś w rodzaju globalnego „rozdawacza" uprawnienia. Proces sięga po uprawnienie tylko wtedy, gdy jest mu ono potrzebne. Zatem przed wejściem do sekcji krytycznej każdy proces powinien wykonać INPUT('uprawnienie') zabierając uprawnienie z przestrzeni, a po wyjściu — OUTPUT('uprawnienie') oddając je z powrotem.

Uważny czytelnik zauważy od razu, że implementacja w Lindzie semafora binarnego (tylko jedna krotka ('sem') w przestrzeni) i przekazywania uprawnienia niczym się od siebie nie różnią.

7.2.2 Producenci i konsumenci

Jak już wcześniej zauważyliśmy, sama przestrzeń krotek odgrywa rolę bufora. Zatem w Lindzie procesy producenta i konsumenta zapisuje się banalnie.

```
const P = ?;   {liczba producentów}
      K = ?;   {liczba konsumentów>

process PRODUCENT(i:l..P);
var p:  porcja;
begin
  while true  do  begin
    produkuj(p);
    OUTPUT(p)
  end
end;

process KONSUMENT(i:l..K);
var p:porcja;
begin
  while true  do  begin
    INPUT(p:porcja);
    konsumuj(p)
  end
end;
```

Jeśli zależy nam na tym, aby wyprodukowane porcje były konsumowane w takiej kolejności, w jakiej je wyprodukowano, musimy sami o to zadbać numerując każclą porcję. Przetrzeń krotek nie gwarantxije nam żadnej określonej kolejności ich pobierania. Jeśli w systemie działa tylko jeden producent i jeden konsument, wystarczy w procesie producenta do krotki z porcją dołożyć mimer porcji j liczony lokalnie przez producenta, i podobnie zrobić to w procesie konsumenta. Jeśli jest wielu prodxicentow i wielu konsumentów, w przestrzeni trzeba umieścić dwa liczniki. Każdy producent (konsument) musi pobrać licznik, skorzystać z jego wartości a następnie wysłać w przestrzeń wartość licznika zwiększoną o jeden. Licznik odgrywa tu rolę semafora wstrzymującego procesy. Początkowo liczniki wskazują na pierwszą porcję, a więc w przestrzeni trzeba umieścić krotki ('lprod' , 1), ('lkons', 1).

```
process PRODUCENT(i:l..P);
var p:  porcja;
    j:  integer;
begin
  while true do begin
    produkuj(p);
    INPUT('lprod', j:integer);
    OUTPUT(p, j);
    OUTPUT('lprod', j+l)
  end
end;

process KONSUMENT(i:l..K);
var p:  porcja;
    j:  integer;
begin
  while true do begin
    INPUT('lkons', j:integer);
    INPUT(p:porcja, j);
```

```
      OUTPUT('lkons', j+l);
      konsumuj(p)
    end
  end;
```

W tym rozwiązaniu mamy w rzeczywistości biifor nieograniczony — producent nie jest nigcly wstrzymywany. Jeśli liczba porcji znajchijących się w jednej cliwili w przestrzeni ma być ograniczona, należy początkowo w przestrzeni umieścić odpowiednią liczbę krotek informujących o wolnych miejscach. Proces PRODUCENT może wyslać porcję w przestrzeń, jeśii wcześniej pobierze wolne miejsce. Proces KONSUMENT po odebraniu porcji powinien dodatkowo zwracać wolne miejsce.

```
process PRODUCENT(i:l..P);
var p:  porcja; j:integer;
begin
  while true do begin
    produkuj(p);
    INPUT('miejsce');
    INPUT('lprod', j:integer);
    OUTPUT(p,j);
    OUTPUT('lprod',j+l)
  end
end;
process KONSUMENT(i:l..K);
var p:porcja;  j:integer;
begin
   while true do begin
     INPUT(>lkons', j:integer);
     INPUT(p:porcja, j);
     OUTPUT('lkons', j+l);
     OUTPUT('miejsce');
     konsurauj(p)
   end
end;
```

Rozwiązanie to przypomina nieco rozwiązanie za pomocą zwykłych semaforów przedstawione w p. 3.2.2. Tutaj jednak nie jest potrzebne wzajemne wykluczanie przy dostępie do bufora. Nawet jeśli dwóch producentów będzie chciało jednocześnie umieścić porcję w przestrzeni, zostaną one przyjęte. Krotka ('miejsce') spełnia tu rolę semafora wolne. Semafor pełne nie jest także potrzebny, gdyż jego rolę spełniają krotki z porcjami.
Samo ich istnienie oznacza, że bufor jest niepusty.

7.2.3 Czytelnicy i pisarze

Rozwiązanie z możliwością zagłodzenia pisarzy

W przestrzeni będzie przechowywana krotka z liczbą aktiialnie czytających czytelników. Każdy czytelnik przed rozpoczęciem czytania będzie pobierał tę krotkę i zwracał krotkę z liczbą o jeden większą. Podobnie po zakończeniu czytania będzie pobierał tę krotkę i zwracał krotkę z liczbą o jeden mniejszą. Pisarz może rozpocząć pisanie, jeśli pobierze krotkę z liczbą 0 oznaczającą, że nikt nie czyta. Krotkę tę przetrzymuje u siebie aż do zakończenia pisania uniemożliwiając w ten sposób rozpoczęcie czytania czytelnikom oraz rozpoczęcie pisania innym pisarzom.

Na początku trzeba w przestrzeni umieścić krotkę z wartością 0.

```
const C = ?;   {liczba czytelników>
      P = ?;   {liczba pisarzy}

processCZYTELNIK(i:  l..C);
var c: integer;
begin
    while true  do begin
       wlasne_sprawy;
       INPUT(c:integer);
       OUTPUT(c+l);
       czyta;
       INPUT(c:integer);
       OUTPUT(c-l)
    end
 end;

 processPISARZ(i:  l..P);
 begin
   while true  do begin
     wlasne_sprawy;
     INPUT(0);
     pisze;
     OUTPUT(0)
   end
 end;
```

Rozwiązanie z możliwością zagłodzenia czytelników

W tym rozwiązaniu krotka, z której korzystają procesy, jest parą liczb. Pierwsza z nich ma takie samo znaczenie jak w poprzednim rozwiązaniu. Druga oznacza liczbę pisarzy oczekujących na pisanie. Czytelnik może rozpocząć czytanie tylko wtedy, gdy pobierze krotkę mającą na drugiej pozycji wartość 0 (żaden pisarz nie czeka). Gdy mu się to uda, zwraca krotkę ze zwiększoną liczbą procesów czytających. Po zakończeimi czytania pobiera krotkę z przestrzeni i zwracają z pierwszą liczbą zmniejszoną o jeden. Pisarz przed rozpoczęciem pisania pobiera krotkę i bada, czy pierwsza liczba jest równa 0 (nikt nie czyta). Jeśli nie, to zwraca krotkę ze zwiększoną drugą liczbą (czyli zgłasza, że jest), a następnie oczekuje na krotkę mającą na pierwszej pozycji wartość 0 (na pewno się doczeka, bo żaden nowy czytelnik nie możejuż rozpocząć czytania). W rozwiązaniu z możliwościązagłodzenia pisarzy pisarz zwracał krotkę do przestrzeni dopiero po zakończeniu pisania. W ten sposób jednak niemożliwa byłaby rejestracja nowych pisarzy, gdy jeden z nich pisze. W rezultacie, po zakończeniu pisania nowo przybyli pisarze musieliby współzawodniczyć o krotkę z czytelnikami, co oznaczałoby, że pisarze nie mają tu priorytetu. (Konsekwencje tego podajemy przy omawianiu rozwiązania poprawnego.) Dlatego pisarz rozpoczynający pisanie będzie zwracał krotkę umieszczając na pierwszej pozycji wartość niezerową, np. —1, co uniemożliwi wejście do czytelni zarówno innym pisarzom, jak i czytelnikom, ale pozwoli rejestrować się nowym pisarzom. Po zakończeniu pisania pisarz będzie zabierał tę krotkę z przestrzeni zastępując ją krotką z zerem na pierwszej pozycji i liczbą czekających pisarzy na drugiej.

Na początku trzeba w przestrzeni umieścić krotkę (0,0).

```
processCZYTELNIK(i: l..C);
var c,p:  integer;
begin
```

```
  while true do blgin
    wlasne_sprawy;
    INPUT(c:integer, 0);
    OUTPUT(c+l, 0);
    czyta;
    INPUT(c:integer, p:integer);
    OUTPUT(c-l, p)
  end
end;

processPISARZ(i: l..P);
var c,p:  integer;
begin
 while true  do begin
   wlasne_sprawy;
   INPUT (c:integer, p:integer);
   if  c > 0  then begin   {trzeba poczekać}
     OUTPUT(c,  p+l);     {zgłoszenie się}
     INPUT(0,  p:integer);
     OUTPUT(-1,  p)        {zwrot krotki>
   end else
     OUTPUT(-1,  p+l);     {zwrot krotki i zgloszenie}
   pisze;
   INPUT(-1, p:integer);
   OUTPUT(0, p-l)         {odmeldowanie się}
 end
end;
```

Rozwiązanie poprawne

W rozwiązaniii z możliwością zagłodzenia czytelników wymagaliśmy, aby pisarz zwracał krotkę do przestrzeni przed rozpoczęciem pisania, umożliwiając w tym czasie rejestrację nowych pisarzy. Gdyby PISARZ zachowywał krotkę na czas pisania, jego treść wyglądałaby następująco:

```
while true do begin
   wlasne_sprawy;
   INPUT(c:integer,  p:integer);
   if c > 0 then  begin  {trzeba poczekać}
      OUTPUT(c, p+l);   {zgłoszenie się}
       INPUT(0, p:integer);
       p  :=  p  - 1         {odmeldowanie  się}
    end;
    pisze;
    OUTPUT(0,  p)          {zwrot  krotki}
 end
```

W tym rozwiązaimi pisarze mogą się rejestrować tylko podczas czytania. Pisarze przybyli w trakcie pisania nie rejestrują się, lecz po prostu czekają, na krotkę z zerem na pierwszej pozycji. W rezultacie, jeśli inicjatywę przejmą pisarze, to po pewnym czasie wszyscy ci, którzy zarejestrowali się podczas czytania, wejdą do czytelni i zmniejszą drugi element krotki do zera. Od tej chwili w przestrzeni będzie się już pojawiać tylko krotka (0,0). Z żywotności operacji INPUT wynika, że w końcu krotkę tę pobiorą czekający czytelnicy. Zaraz potem zarejestrują się czekający pisarze i po pewnym czasie oni przejmą inicjatywę. Nie jest więc tu możliwe zagłodzenia ani pisarzy, ani czytelników. W przedstawionym rozwiązaniu nie można określić kolejności wchodzenia procesów do czytelni (wynika ona jedynie ze sposobu realizacji operacji INPUT).

Omówimy jeszcze rozwiązanie, w którym czytelnicy i pisarze pragnący jednocześnie wejść do czytelni wchodzą do niej na przemian (pisarze pojedynczo, a czytelnicy w grupie). Użyjemy trzech rodzajów krotek: krotka o sygnaturze (integer, integer) ma takie samo znaczenie jak poprzednio, o sygnaturze (integer) służy do zliczania czekających czytelników, krotka ('start_czyt') zaś służy do wznawiania czytelników. Czytelnik próbuje dołączyć do innych czytelników wtedy, gdy nie ma czekających pisarzy. Gdy mu się to nie uda, zwiększa licznik czekających czytelników i oczekuje na krotkę ('start_czyt'), która umożliwi mu rozpoczęcie czytania. Pisarz przed wejściem do czytelni zachowuje się tak samo, jak w poprzednim rozwiązaniu. Natomiast po wyjściu z niej pobiera krotkę z liczbą czekających czytelników i wprowadza do przestrzeni krotek odpowiednią liczbę krotek postaci ('start_czyt'). Następnie zeruje licznik czekających czytelników i zwraca krotkę z nową liczbą czytających czytelników. W ten sposób po każdym zakoiiczenhi pisania do czytelni jest wpuszczana grupa czytelników, którzy zebrali się w czasie tego pisania.

```
Process CZYTELNIK(i:  l..C);
 var c,p: integer;
     cc:  integer;   {czytelnicy  czekający}
begin
    while true do begin
      wlasne_sprawy;
      if TRY_INPUT(c:integer, 0)  then OUTPUT(c+l,  0)
     else  begin
        INPUT(cc:integer);
        OUTPUT(cc+l);
        INPUT('start_czyt')
     end;
     czyta;
     INPUT(c:integer, p:integer);
     OUTPUT(c-l, p)
   end
end;

process PISARZ(i: l..P);
var c,p: integer;
    cc: integer;   {czytelnicy  czekający}
begin
  while true  do begin
    wlasne_sprawy;
    INPUT(c:integer, p:integer);
    if  c > 0  then begin
      OUTPUT(c, p+l);
      INPUT (0, p:integer);
      p  := p  - 1
    end;
    pisze;
    INPUT  (cc:integer);
    for i  :=  1 to cc do OUTPUT  ('start_czyt');
    OUTPUT (0);
    OUTPUT (cc, p)
  end
end;
```

Rozwiązanie to kryje.w sobie pewne niebezpieczeństwo. Jeślijakiś czytelnik wykonaTRY_INPUT z sukcesem, ajeszcze nie wykona OUTPUT, to następny czytelnik wykonujący właśnie w tym czasie TRY_INPUT stwierdzi, iż krotki nie ma i zarejestruje się jako czytelnik czekający (będzie mu się wydawać, że jest jakiś pisarz). Można to poprawić czyniąc operację TRY_INPUT sekcją krytyczną procesu. Wymaga to wprowadzenia do przestrzeni

dodatkowej krotki ('sprawdzanie') pełniącej funkcję semaforabinarnego. Odpowiedni fragment procesu CZYTELNIK powinien wówczas wyglądać następująco:

```
INPUT('sprawdzanie');
if TRY_INPUT  (c:integer, 0) then begin
   OUTPUT  (c+l, 0);
   OUTPUT('sprawdzanie')
 end else begin
   OUTPUT('sprawdzanie');
   INPUT  (cc:integer);
```

Uważny czytelnik spostrzeże (tym razem to chodzi o Ciebie, Drogi Czytelniku!), że znacznie prościej jest użyć krotki trójelementowej zawierającej liczbę czytających czytelników, liczbę czekających pisarzy i liczbę czekających czytelników. Nie będzie wówczas potrzebna operacja TRY_INPUT. Na tym przykładzie chcieliśmy jednak pokazać, jakie niebezpieczeństwa wiążą się z używaniem tej operacji, stąd ten nieco zagmatwany algorytm. Czytelnikowi pozostawiamy zapisanie prostszego rozwiązania.

7.3 Zadania

7.3.1 Zasoby dwóch typów

Wersja 1. Zapisz w Lindzie rozwiązanie zadania 4.3.10. Przyjmij, że zasoby są pasywne.
Wersja 2. Zapisz w Lindzie rozwiązanie zadania4.3.10. Przyjmij, że zasoby są aktywne, czyli są współdziałającymi procesami, oraz że są tylko dwa typy procesów użytkownika: korzystające tylko z zasobu typu A i korzystające z dowolnego typu zasobu. Proces korzystający z zasobu przesyła mu do przetworzenia porcję, a w wyniku otrzymuje przetworzoną porcję.

7.3.2 Obliczanie histogramu

Zapisz w Lindzie rozwiązanie zadania 5.3.3. Procesy SEG powinny być napisane w ten sposób, aby obliczenie histogramu można było zrealizować uruchamiając dowolną ich liczbę, niekoniecznie równą liczbie wartości progowych. Dla ustalenia uwagi załóż, że układ procesów obh'cza histogram dla wartości funkcji sinus w punktach (-10,-9,...,9,10) przy wartościach progowych (-0.8,-0.4,0.0,0.4,0.8,1.2).

7.3.3 Głosowanie

Zapisz w Lindzie rozwiązanie zadania 5.3.5. Do rozsyłania glosów zastosuj przestrzeń krotek i możliwość rozgłaszania dzięki użyciu operacji READ. Zastanów się, jak można zapewnić, aby po zakończeniu głosowania przestrzeń krotek była pusta.

7.3.4 Komunikacja przez pośrednika

Zapisz w Lindzie rozwiązanie zadania 5.3.6. Zamiast procesu POŚREDNIK użyj przestrzeni krotek.

7.3.5 Centrala telefoniczna

Zapisz w Lindzie rozwiązanie zadania 5.3.7. Zamiast procesu CENTRALA użyj przestrzeni krotek.

7.3.6 Mnożenie wielomianów

Zapisz w Lindzie algorytm mnożenia wielomianów podany w zadaniu 5.3.12. Procesy P powinny być napisane w ten sposób, aby mnożenie wielomianów można bylo zrealizować uruchamiając dowolną ich liczbę.

7.3.7 Mnożenie macierzy

Zapisz w Lindzie algorytm współbieżnego obliczania iloczynu dwóch macierzy o rozmiarach N*M i M*K. Każdy z elementów macierzy wynikowej ma być obliczany niezależnie od pozostałych. Proces UŻYTKOWNIK powinien umieszczać w przestrzeni krotek mnożone macierze i pobierać z niej ostateczny wynik. Procesy wykonujące obliczenia powinny być tak napisane, aby mnożenie macierzy można bylo wykonać uruchamiając dowolną ich liczbę.

(Zadanie to dla przypadku macierzy 3 * 3 zostało opisane i rozwiązane w [BenA90].)

7.3.8 Problem ośmiu hetmanów

Problem ośiimi hetmanów formułuje się następująco: ustawić na szachownicy osiem hetmanów tak, aby się nawzajem nie szachowały (por. [Wirt80], rozdział 3.5). Jest to typowy problem, który rozwiązuje się metodą prób i błędów zwaną metodą nawracania. W programach sekwencyjnych metoda ta polega na stopniowym budowaniu końcowego rozwiązania w zgodzie z narzuconymi warunkami (w tym przypadku dostawianiu kolejnego hetmana tak, aby nie szachowałjuż stojących) i odpowiednim wycofywaniu się, gdy uzyskanego częściowego rozwiązania nie możnajuż dalej rozszerzać. W programach współbieżnych równolegle bada się i rozszerza wiele rozwiązań częściowych, a te, których dalej nie można już rozszerzać, po prostu się porzxica.

Zapisz w Lindzie algorytm współbieżnego rozwiązania problemu ośmiu hetmanów. Przyjmij, że częściowe rozwiązanie jest pamiętane w ośmioelementowej tablicy, przy czym i-ty element tej tablicy jest pozycją hetmana w i-tej kolumnie szachownicy. Załóż dlauproszczenia, żejest dostępnafunkcja logiczna proba(i,j ,uklad) odpowiadająca na pytanie, czy hetmana można postawić na pozycji (i,j),jesli położenia pozostałych hetmanów znajdujących się w pierwszych j-1 kolumnach są podane w tablicy układ. Algorytm powinien podawać wszystkie możliwe ustawienia hetmanów lub stwierdzać, że ich nie ma. Procesy badające ustawienia powinny być tak napisane, aby problem można było rozwiązać uruchamiając dowolną ich liczbę.

7.3.9 Obliczanie całki oznaczonej

Numeryczne wyznaczanie wartości całki oznaczonej pewnej dodatniej ciągłej funkcji f(x) polega na oszacowaniu pola powierzchni pod wykresem tej funkcji. Jedną z metod jest przybliżanie tego pola polami trapezów. Aby wyznaczyć całkę na odcinku [a, b] z dokładnością e trzeba porównać pole trapezu o wierzchołkach (o,0), (&,0), (a,/(o)) i (6,/(6)) z sumą pól dwóch trapezów o wierzchołkach (a,0), (c,0), («,/(«)), (c,/(c)) oraz (c,0), (6,0), (c,/(c)), (b,f(b)), przy czym c jest tu środkiem odcinka [a,b], (c = (a + 6)/2). Jeśli wartość bezwzględna różnicy nie przekracza £, za wartość całki przyjmujemy pole pojedynczego trapezu. Jeśli różnicajest większa, wyznaczamy całki oddzielnie na przedziałach [a, c] i [c, b] z dokładnością s/'2 i wyniki sumujemy. Wyznaczanie ołm tych calek może odbywać się równolegle. Zapisz w Lindzie algorytm współbieżnego wyznaczania całki oznaczonej. Procesy wykonujące obliczenia powinny być tak napisane, aby całkę można bylo obliczyć uruchamiając dowolną ich liczbę.

(Zadanie to zostało omówione i rozwiązane w [Andr91].)

7.4 Rozwiązania

7.4.1 Zasoby dwóch typów

Wersja 1. Każdy zasób będzie reprezentowany w przestrzeni krotek przez krotkę zawierającą jego typ i numer. Pobranie krotki będzie oznaczać uzyskanie prawa dostępu do zasobu. Po zakończeniu korzystania z zasobii proces będzie zwracał krotkę do przestrzeni. Proces INICJATOR wprowadzi na początku odpowiednie krotki do przestrzeni. Proces grupy drugiej próbuje najpierw pobrać krotkę zasobu wygodniejszego, a gdy to się nie uda, czeka na dowolną krotkę. Proces grupy trzeciej próbuje pobrać dowolną krotkę i, jeśli się to uda, korzysta z zasobu.

```
    const M =  ?;    {liczba  zasobów typu  A}
          N =  ?;    {liczba  zasobów typu  B}

    process INICJATOR;
    var i: integer;
begin
  for i := 1 to M do
    OUTPUT('A', i);
  for i := 1 to N do
    OUTPUT('B', i)
end;

process GRUPA1;
var i: integer;
begin
  while true do begin
    wlasne_sprawy;
    INPUT('A', i:integer);
    korzystaj('A',i);
    OUTPUT('A',i)
  end
end;

process GRUPA2;
type zasób = 'A'..'B';
```

```
var i: integer;
    jaki:  zasób;
begin
  while true do begin
    wlasne_sprawy;
    if TRY_INPUT('A', i:integer)  then
      jaki :=  'A'
    else INPUT(jaki:zasob,  i:integer);
    korzystaj(jaki, i);
    OUTPUT(jaki, i)
  end
end;

process GRUPA3;
type zasób = 'A'..'B';
var i: integer;
    jaki:  zasób;
begin
  while true do begin
    wlasne_sprawy;
    if TRY_INPUT('B', i:integer)
    then begin
         korzystaj('B',i);
         OUTPUT('B',i)
       end  else
       if TRY_INPUT(jaki:zasob,  i:integer)
       then begin
         korzystaj(jaki,  i);
         OUTPUT(jaki, i)
       end
     end
   end;
```

Proces GRUPA3 najpierw próbuje pobrać krotkę zasobu mniej wygodnego, a jeśli to się nie nda, próbuje pobrać jakąkolwiek krotkę. To wyróżnienie zasobu typu B ma na celu zmniejszenie obciążenia zasobów typu A, które są używane tylko przez proces GRUPA1. Zauważmy, że gdy GRUPA3 stwierdzi, że nie ma krotki zasobu typu B, wykonuje operację TRY_INPUT bez wskazywania typu zasobu, może bowiem się tak zdarzyć, że między dwoma kolejnymi operacjami TRY_INPUT pojawi się nowa krotka zasobu typu B. Wersja 2. Procesy drugiej grupy wysyłają w przestrzeń krotki z porcjami nie wskazując zasobu, który ma je przetworzyć.

```
   process  A(i: 1..M);
   var p: porcja;
       j: integer;
   begin
     while  true do begin
       INPUT('A',  j:integer,  p:porcja);
       przetwórz(p);
       OUTPUT(j, p)
     end
   end;

   process  B(i:1..N);
   var p: porcja;
       j: integer;
   begin
     while  true do begin
       INPUT('B',  j:integer,  p:porcja);
       przetwórz(p);
```

```
      OUTPUT(j, p)
    end
  end;
  process GRUPA1;
  var p:  porcja;
  begin
    while true do begin
      wlasne_sprawy(p);
      OUTPUT('A', 1, p);
      INPUT(1, p:porcja)
    end
  end;

  process GRUPA2;
  type  zasób = 'A'..'B';
  var p:  porcja;
      jaki:  zasób;
  begin
    while true do begin
      wlasne_sprawy(p);
      OUTPUT(jaki:zasob,  2,  p);
      INPUT(2, p:porcja)
    end
  end;
```

Jeśli procesów obu grup jest więcej niż po jednym, każdy z nich musi dodatkowo przekazywać swój jednoznaczny identyfikator, aby na jego podstawie mógł następnie pobrać z przestrzeni zwróconą mu przetworzoną porcję.

7.4.2 Obliczanie histogramu

W następującym rozwiązaniu mamy pięć rodzajów krotek: o sygnaturze (string, integer), na podstawie której proces SEG określa swój numer, o sygnaturze (string, integer, real) z wartością progową o podanym numerze, o sygnaturze (integer, real) z wartościami funkcji skierowanymi do odpowiedniego procesii, o sygnaturze (integer) z sygnałami do procesu SEG od wskazanego procesu oraz krotka z ostatecznym wynikiem umieszczonym w tablicy typu tab. Przyjmujemy, że sygnałem końca danych dla procesów SEG będzie wartość większa niż wartość największego progu, a sygnałem końca danych dla procesu STAT będzie liczba większa niż n-czba wartości progowych. Po otrzymaniu sygnału końca danych proces ma pewność, że wszystkie przeznaczone dla niego krotki znajdują się już w przestrzeni. Może je więc pobierać do skutku operacją TRY_INPUT.

```
   const  N =  6;               {liczba  wartości  progowych}
process SEGO;
type tab = array[l..N]  of integer;
var i,j:  integer;
    h: tab;
begin
  OUTPUT('numer',  1);
  for i := 1 to N  do       {wysłanie progów}
    OUTPUT('prog', i,  (i-3)*0.4);
  for i := -10 to  10 do    {wysłanie wartości}
    OUTPUT(1, sin(i));
  OUTPUT(1, 1.3);           {sygnal  końca}
  INPUT(h:tab)             {odebranie wyniku}
end;
```

```
process SEG;
var i: integer;
    a,araax,x: real;
    jeszcze: boolean;
begin
  repeat
    INPUT('numer', i:integer);
                          {odczytanie numeru}
    jeszcze := i <= N;
    ifjeszcze then        {odczytanie maksymalnego progu}
      READ('prog', N,  amax:real);
    OUTPUT('numer', i+l);  {wysłanie numeru następnemu}
    if jeszcze then begin
      INPUT('prog', i,  a:real);
                          {wczytanie własnego progu}
      repeat
        INPUT(i, x:real);  {wczytanie wartości}
        if x < a then      {dobra wartość}
          OUTPUT(i)
        else if x  < amax then
          OUTPUT(i+l,x)    {przekazanie dalej}
      until x > amax;       {doszedł sygnał końca}
      while TRY_INPUT(i, x:real) do begin
        INPUT(i, x:real);  {wczytanie wartości}
        if x < a then      {dobra wartość}
          OUTPUT(i)
        else OUTPUT(i+l,x)  {przekazanie dalej}
      end;
          if i = N then        {sygnał końca dla STAT>
            OUTPUT(N+1)
          else                  {sygnał końca dla SEG}
            OUTPUT(i,  amax+l)
        end
     until not jeszcze
   end;

   process STAT;
   var  h:  array[l..N]  of integer;
        i:  integer;
   begin
     for i := 1 to  N  do h[i]  :=  0;
     repeat
        INPUT(i:integer);
        if  i <= N then h[i]  := h[i]  + 1
     until i > N;              {odebrano sygnał  końca}
     while TRY_INPUT(i:integer)  do
        h[i] := h[i] + 1;
     OUTPUT(h)
   end;
```

W tym rozwiązaniu każdy nowo uruchomiony proces SEG dowiaduje się na początku, jaki ma numer. Jeśli numer ten jest większy niż N, to proces wysyla w przestrzeń liczbę o jeden większą i kończy działanie. W przeciwnym razie wczytuje wartość progową oraz liczby, które w przestrzeni umieścił dla niego proces o numerze o jeden mniejszym, i wykonuje swój właściwy algorytm. Po otrzymaniu sygnału końca, czyli liczby większej niż nmax, proces SEG wczytuje krotki operacją TRY_INPUT, aby rozpoznać koniec danych. Po wczytanhi wszystkich danych wysyła sygnał końca kolejnemu procesowi. Następnie wczytuje nowy numer procesu (który wcale nie musi być numerem o jeden większym od poprzedniego, albowiem w tym czasie mogły zostać uruchomione inne procesy SEG), po czym wykonuje

algorytm procesu SEG o wczytanym numerze. Jak widać, system będzie działał poprawnie niezależnie od tego, czy współbieżnie uruchomi się jeden, dwa, czy sześćprocesów SEG.

7.4.3 Głosowanie

W odróżnieniu od rozwiązania w CSP z p. 5.4.5 nie potrzebujemy procesu SIEĆ, ale także nie trzeba wyróżniać procesu pomocniczego LICZ. Nie ma tu bowiem niebezpieczeństwa powstania blokady, gdyż procesy nie synchronizują się ze sobą przy przekazywaniu głosów. Podany niżej proces P powstał przez włączenie do procesu P z p. 5.4.5 instrukcji liczenia glosów. Głosy odczytuje się za pomocą operacji READ, aby umożliwić odczytanie tego samego glosu także innym procesom. Przy odczytywaniu wskazuje się numer głosującego i numer kolejny tury.

```
const N = ?;  {liczba procesów}

process P(i:l..N);
var T: array[l..N]  of integer;
    j,min,tura:  integer;
    koniec: boolean;
begin
  for j := 1 to  N do T[j] :=  1;
  tura := 0;
  repeat
    tura := tura +  1;
    OUTPUT(i, tura,  glosuj(T));
    for j := 1 to N  do T[j]  := 0;
    for j := 1 to N  do begin
      READ(j, tura,  k:integer);
      T[k]  := T[k]  + 1
    end;
    min := 1;              {szukanie niezerowego minimum}
    while T[min]  =  0 do min  := min  + 1;
    for j := min+1  to N do
      if T[j] >  0 then
      if T[j] <  T[min] then min  :=  j;
    koniec := T[min] = N;
    T[min]  := 0
  until koniec
end;
```

Po zakończeniu głosowania w przestrzeni zostaną krotki opisujące głosy każdego procesu w każdej turze. Moglibyśmy zażyczyć sobie, aby ostatni proces odczytujący głos usuwał krotkę z przestrzeni operacją INPUT. Niestety proces nie wie, czyjest ostatnim odczytującym dany głos. Potrzebnabyłaby więc dodatkowa synchronizacja. Można również wymagać, aby krotkę z glosem usuwał ten proces, który wyslał ją w przestrzeń. Wolno mu to jednak zrobić dopiero wtedy, gdy ma już pewność, że wszystkie inne procesy odczytały jego glos. Pewność tę proces może mieć jednak dopiero wtedy, gdy zbierze wszystkie głosy z następnej tury. Tak więc na końcu pętli while można umieścić instrukcję:

```
if tura > 0 then  INPUT(i, tura-l, j:integer)
```

Pozostanie nadal problem usunięcia krotki z ostatnim glosem każdego procesu. Jest potrzebna dodatkowa synchronizacja procesów po zakończeniu głosowania. (Przykład takiej synchronizacji omawiamy w rozwiązaniu 7.4.8.)

Wymienionych problemów można uniknąć, jeśli każdy proces będzie wysyłał w przestrzeń swój glos nie raz, ale N razy, a glosy będą wczytywane operacją INPUT. Rozwiązanie takie wymaga jednak aż N2 operacji OUTPUT. Inna możliwość to rozszerzenie krotki o czwartą pozycję oznaczającą liczbę odczytów (początkowo równą N) i używanie operacji INPUT zamiast READ.

Każdy proces po wczytaniu krotki zmniejszałby liczbę odczytów i zwracał krotkę, gdyby liczba odczytów byla, jeszcze dodatnia. Takie rozwiązanie z kolei zmniejszałoby równoległość działań — czas odczytania krotki bylby sumą czasów potrzebnych na jej odczytanie przez każdy proces.

7.4.4 Komunikacja przez pośrednika

Zapisanie w Lindzie rozwiązania tego zadania okazuje się banalne. Na początku musimy umieścić w przestrzeni M krotek z komunikatami. Robi to proces INICJATOR używając procedury nowy_komunikat.

```
const M =  ?;  {liczba komunikatów}
      N =  ?;  {liczba procesów}

process INICJATOR;
var i,j:  integer;   k:  komunikat;
begin
   for i := 1 to M  do begin
    losuj(j) ;
    k  := nowy_komunikat;
    OUTPUT(j, k)
   end
end;

process P(i: l..N);
var j:  integer;  k:  komunikat;
begin
  while true do begin
    INPUT(i, k:komunikat);
    przetwórz(k);
    losuj(j) ;
    OUTPUT(j, k)
   end
end;
```

7.4.5 Centrala telefoniczna

Przestrzeń krotek musimy odpowiednio zainicjować wstawiając do niej na początku informacje o dostępnych łączach. Nadawca pobiera numer łącza z przestrzeni, wysyła koimmikat z tym numerem do wylosowanego odbiorcy i oczekuje na potwierdzenie odbioru (podniesienie słuchawki). Następnie nadaje, po czym zwalnia łącze. Odbiorca cyklicznie przyjmuje zgłoszenie, potwierdza jego odbiór, a następnie odbiera informację. Podczas nawiązywania połączenia odbiorca nie wie, kto do niego dzwoni. Potwierdzenie odbioru wysyła wraz z numerem otrzymanego lącza. Numer ten jest konieczny do rozróżnienia potwierdzeń w sytuacji, gdy kilku nadawców usiłuje nawiązać połączenie z tym samym odbiorcą.

```
const N  = ?;  {liczba  nadawców}
```

```
      M  = ?;  {liczba  odbiorców}
      K  = ?;  {liczba  łączy}

process  INICJATOR;
var k:  integer;
begin
  for k  := 1 to K do
    OUTPUT('lacze',k)
end;

process  NADAWCA(i:l..N);
var k,j:  integer;
begin
  while  true do begin
    INPUT('lacze', k:integer);
    losuj(j);
    OUTPUT('dzwoni', k, j);
    INPUT('odbior', k,  j);
    nadaj(k);
    OUTPUT('lacze', k)
  end
end;

process  ODBIORCA(i:l..M);
var k:  integer;
begin
  while  true do begin
    INPUT('dzwoni', k:integer,  i);
    OUTPUT('odbior', k, i);
    odbierz(k)
  end
end;
```

7.4.6 Mnożenie wielomianów

Proces UŻYTKOWNIK wysyła w przestrzeń współczynniki mnożonych wielomianów, po jednej krotce ('iloczyn') na każcly iloczyn do obliczenia oraz numery przekątnych, wzdłuż których mają, przebiegać obliczenia. Każdy proces P próbuje wczytać na początku krotkę ‚^loczynowa". Jeśli mu się to nie uda, kończy działanie, bo wszystkie iloczyny zostały już policzone. W przeciwnym razie próbuje rozpocząć obliczenia na nowej przekątnej lub kontynuować już rozpoczęte. Współczynniki wielomianów są odczytywane z przestrzeni operacją READ, bo mogą być jeszcze potrzebne dla obliczenia innych iloczynów, sumy częściowe zaś pobiera się operacją INPUT.

```
const  N =  ?;                    {stopień mnożonych
                                  wielomianów}
process UŻYTKOWNIK;
var a,b: array[O..N} of real; {mnożone wielomiany}
    i,j: integer; x: real;
    c:  array[0..2*N] of real; {wielomian wynikowy}
begin
                                {ustalenie wartości
                                  a[0. .N]  i b[0. .N]}
  for  i := 0 to  N do begin
    OUTPUT('a',  i,  a[i]);     {wysłanie  ich w przestrzeń}
    OUTPUT('b',  i,  b[i])
  end;
  for  i := -N to N do
```

```
      OUTPUT(i);                  {numery przekątnych}
    for  i := 1  to  (N+1)*(N+1)  do
      OUTPUT('iloczyn');         {krotki iloczynowe}
    for  i := -N to N do begin
      INPUT(j:integer, x:real); {odbiór współczynników}
      c[j]  := x                  {iloczynu}
    end;
    for  i := 0 to  N do begin
      INPUT('a',  i, a[i]);      {oczyszczenie przestrzeni}
      INPUT('b',  i, b[i])
    end;
                                  {korzystanie z wyniku}
  end;
  process P;
  var  a,b,c: real;
       i,j.k: integer;
  begin
     while TRY_INPUT('iloczyn')  do
       if TRY_INPUT(k:integer) then
       begin                        {nowa przekatna}
         c := 0;                    {suma częściowa}
         if k < 0  then begin       {ustalenie  współrzędnych}
           j  := N  +  k;
           i  := 0
         end else  begin
           j  := N,
           i  := k
         end
       end else                     {kontynuacja przekątnej}
         INPUT('c',  i:integer, j:integer,  c:real);
       READ('a', i,  a:real);
       READ('b>, j,  b:real);
       if j * (i-N)  = 0 then       {koniec przekątnej}
         OUTPUT(i+j,  c+a*b)
       else                         {dane dla następnego}
         OUTPUT('c',  i+l, j-l, c+a*b)
     end
  end;
```

Warto zauważyć, że proces UŻYTKOWNIK wysyla do przestrzeni najpierw numery przekątnych, a potem krotki „iloczyiiowe". Gdyby robil to odwrotnie, procesy P mogłyby stwierdzić, że nie ma już żadnej przekątnej do rozpoczęcia, i oczekiwać na wykonanie operacji INPUT. Jeśli wszystkie znalazłyby się w tym stanie, mielibyśmy typowy przykład blokady.

Prezentowane rozwiązanie ma tę własność, że po zakończeniu obliczania iloczynu przestrzeń krotek jest pusta, a procesy P kończą się poprawnie. (W tym przypadku jest to możliwe, gdyż znamy liczbę operacji częściowych składających się na pełne obliczenie — por. 7.4.8.) Efekt ten uzyskano kosztem wprowadzenia do przestrzeni (N+ 1)2 krotek „iloczynowycli". Gdyby zamiast nich w przestrzeni umieścić pojedynczy licznik, liczba koniecznych operacji wejścia-wyjścia nie zmieniłaby się, ale przestrzeń byłaby znacznie luźniejsza. Licznik musiałby być jednak wczytywany operacją INPUT i powstałby problem, kto i kiedy ma go usunąć, aby oczyścić przestrzeń po zakończeniu obliczeń. Bez względu na to, jaką damy odpowiedź, jakiś spóźniony proces P mógłby nie zdążyć ze stwierdzeniem, że obliczenia są już skończone, i zacząć oczekiwać na nieistniejący licznik.

7.4.7 Mnożenie macierzy

Do obliczenia każdego elementu macierzy wynikowej jest potrzebny odpowiedni wiersz pierwszej macierzy i odpowiednia kolumna drugiej, dlatego proces UŻYTKOWNJK umieszcza w przestrzeni krotek wiersze pierwszej macierzy i kolumny drugiej. Proces WYKONAWCA dokonujący obliczeń imisi wiedzieć, który element macierzy wynikowej maliczyć. W tym celu UŻYTKOWNIK umieszcza w przestrzeni specjalną krotkę zawierającą licznik. Początkowo ma on wartość 1 i jest zwiększany przez każdy proces WYKONAWCA. Gdy wartość licznika przekroczy N*K, obliczenia procesu WYKONAWCA kończą się.

```
const  N  = ?;                        {rozmiary macierzy}
       M  = ?;
       K  = ?;

process  UŻYTKOWNIK;
var a: array[l..N,l..M]  of  real;
    b: array[l..M,l..K]  of  real;
    c: array[l..N,l..K]  of  real; {macierz wynikowa>
    pom:  array[l..M] of  real;     {wektor pomocniczy}
    i,j,l:  integer;
    p: real;
begin

  {wypełnienie tablic a  i b>
  OUTPUT(1);                        {wyslanielicznika>
  for  i  :=  1  to N do begin        {wysłanie wierszy}
    for  j  :=  1 to M do
       pom[j]  := a[i,j] ;
    OUTPUT(i, 'wiersz',  pom)
  end;
  for  i  :=  1  to K do begin        {wysłanie kolumn}
    for  j  :=  1 to M do
       pom[j]  := b[j ,i] ;
    OUTPUT(i, 'kolumna', pom)
  end;
  for  1  :=  1  to N*K do begin      {pobranie wyników}
    INPUT(i:integer, j:integer,  p:real);
    c[i,j]  := p
  end;

  {korzystanie z wyniku}
end;
proces WYKONAWCA;
type  wektor = array[l..M] of real;
var w,kol: array[l..M]  of real;
    i,j,l,s: integer;
    c: real;
begin
  while true do begin
    INPUT(l:integer);
    if 1 <= N*K then OUPUT(1+1);
    if 1 <= N*K then begin
       i := (1-1) div N  + 1;
       j := (1-1) mod K  + 1;
       INPUT(i,  'wiersz', w:wektor);
       INPUT(j,  'kolumna', kol:wektor);
       c := 0;
       for s  :=  1 to  M do
         c := c + w[s]*kol[s] ;
```

```
        OUTPUT(i,j,c)
      end
    end
  end;
```

Zauważmy, żeliczba uruchomionych procesów WYKONAWCAjest w tym rozwiązaniu dowolna. Choć obliczają one poszczególne elementy macierzy wynikowej w ustalonej kolejności (wymuszają krotka zawierającalicznik), to swoje wyniki mogą umieszczać w przestrzeni krotek dowolnie. Proces UŻYTKOWNIK pobiera krotki wynikowe ,jak leci"; indeksy towarzyszące obliczonemu elementowi pozwalają umieścić go we właściwym miejscu tablicy wynikowej.

7.4.8 Problem ośmiu hetmanów

Wszystkie częściowe rozwiązania będą pamiętane w przestrzeni krotek. Proces WYKONAWCA pobiera częściowe rozwiązanie i tworzy następne próbując dostawić hetmana w różnych wierszach kolejnej wolnej kolumny. Proces dostawiający hetmana w kolumnie 8 generuje ostateczne rozwiązanie, które odbierze z przestrzeni krotek proces INICJATOR. W celu odróżnienia rozwiązań ostatecznych od częściowych, te pierwsze są umieszczane w przestrzeni razem z numerem kolumny, w której trzeba dostawić hetmana.

```
proces INICJATOR;
type  wektor = array[1..8] of integer;
var układ: wektor;
    i:  integer;
begin
  for  i  := 1 to 8 do  uklad[i] := 0;
  OUTPUT(l, uklad);           {początkowo pusta szachownica}
  while  true do begin        {pobranie wyników}
     INPUT(uklad:wektor);
     wypisz(uklad)
  end
end;

proces WYKONAWCA;
type wektor = array[1..8]  of integer;
var  układ: wektor;  i,j:  integer;
begin
  while  true do begin
     INPUT(j:integer,  uklad:wektor);
     for  i := 1 to 8 do
       if proba(i,j,uklad)  then begin
         uklad[j]  := i;
         if j = 8 then
           OUTPUT(uklad)       {jest pełne rozwiązanie}
         else
           OUTPUT(j+l, układ) {rozwiązanie częściowe}
       end
  end
end;
```

Wydawać by się moglo, że można tu użyć tylko jednego rodzaju krotek z założeniem, że rozwiązaniem ostatecznym jest rozwiązanie częściowe, w którym hetmana trzeba postawić w kolumnie 9. Wówczas proces INICJATOR mógłby pobierać wyniki operacją INPUT(9, układ: wektor). Trzeba by jednak wtedy zabronić procesowi WYKONAWCA pobierania krotek

wynikowych, a niestety nie ma w Linclzie mechanizmu umożliwiającego podanie w operacji INPUT dopuszczalnego zakresu elementii pobieranej krotki.

Powyższy algorytm generuje co prawda wszystkie możliwe rozwiązania problemu ośnmi hetmanów, ale caly system po zakończeniu obliczeń przechodzi w stan blokady. Proces INICJATOR, który pobiera i wypisiije wyniki w pętU nieskończonej, nigdy zatem nie wie, czy wypisał je już wszystkie. Zwłaszcza, jeśli rozwiązanie w ogóle nie istnieje, nie można się o tym przekonać w skończonym czasie.

Z góry nie wiadomo, ile jest rozwiązań, dlatego proces INICJATOR musi być informowany z zewnątrz o zakończeniu obliczeń. Tylko wówczas będzie mial pewność, że wszystkie rozwiązania znajdują się już w przestrzeni krotek. Aby umożliwić mu pobieranie wyników również przed otrzymaniem sygnahi końca, sygnał ten będzie miał taką samą sygnaturę co krotki z wynikami. Wyniki będą umieszczane w przestrzeni wraz z wartością logiczną true, sygnał końca będzie miał wartość logiczną false.

Zakładamy, że w celu rozwiązania problemu ośmiu hetmanów może być uruchomiona dowolnaliczba procesów WYKONAWCA (zależna np. od liczby wolnych procesorów). Muszą one zatem w jakiś sposób rozpoznać zakończenie obliczeń. Wiadomo, żejeśli obliczenie się zakończy, to przestrzeń nie zawiera już krotek do przetworzenia.

Twierdzenie odwrotne nie jest prawdziwe, gdyż brak krotek może oznaczać, że wszystkie one są w trakcie przetwarzania. Aby rozpoznać ten stan, do przestrzeni wprowadzimy dodatkowo licznik procesów aktywnych. Każdy WYKONAWCA na początku zwiększa licznik zgłaszając swoje istnienie. Krotki z rozwiązaniami częściowymi są pobierane operacją TRY_INPUT. W razie niepowodzenia WYKONAWCA zmniejsza licznik wskazując tym samym, że nie ma co robić. Proces, który stwierdzi, iż jest ostatnim nie mającym nic do roboty, wysyła sygnał końca. W przeciwnym razie, ponieważ w przestrzeni mogą pojawić się jeszcze nowe krotki, proces przechodzi do wykonywania operacji READ, której jedynym zadaniem jest ponowne jego wzbudzenie. Jeśli pojawi nowa krotka, operacja READ wykona się i proces ponownie zgłosi swoją aktywność, po czym spróbuje pobrać właśnie przeczytaną krotkę. Krotki tej może jednak już nie być w przestrzeni, gdyż może ją pobrać jakiś inny proces, w szczególności ten, który ją stworzył. Procedura wypisz podaje końcowy uklad hetmanów na urządzeniu wyjściowym (ekran, drukarka).

```
proces  INICJATOR;
type wektor = array[1..8]  of integer;
var układ:  wektor;   i: integer;
     b,  jeszcze; boolean;
begin
  for i :=  1 to 8 do uklad[i] := 0;
  OUTPUT(1,  układ);        {na początku pusta szachownica>
  OUTPUT(0);               {licznik}
  jeszcze  := true;
  while jeszcze do  begin  {nie  było sygnału końca}
     INPUT(jeszcze:boolean, uklad:wektor);
     if  jeszcze wypisz(uklad)
  end;                      {wszystkie  rozwiązania już są}
                            {w przestrzeni,  więc bierzemy}
                            {je,  aż  do  skutku}
  while TRY_INPUT(b:boolean, uklad:wektor) do
     wypisz(uklad)
  end
end;
process  WYKONAWCA;
type  wektor = array[1..8]  of  integer;
var układ:  wektor;  i>j:  integer;
begin
  INPUT(i:integer);
```

```
    OUTPUT(i+l);              {zgłoszenie aktywności}
    while  true do begin
      while not TRY_INPUT(j:integer, uklad:wektor)
      do begin                {w  przestrzeni nic  nie  ma}
        INPUT(i:integer);     {sprawdzenie licznika}
        if i  = 1 then         {sygnał końca}
           OUTPUT(false, układ)
        else  OUTPUT(i-l);     {zgłoszenie nieaktywności}
        READ(j: integer, układ: wektor);
                               {może coś jeszcze się pojawi}
        INPUT(i:integer);
        OUTPUT(i+l)           {ponowne zgłoszenie aktywności}
      end;
      for  i := 1 to 8 do
        if proba(i,j,uklad)  then begin
           uklad[j]  := i;
           if  j = 8 then       {jest pełne rozwiązanie}
             OUTPUT(true, układ)
           else                 {rozwiązanie częściowe}
             OUTPUT(j+l, układ)
        end
    end
  end;
```

Po znalezieniu wszystkich układów procesy WYKONAWCA zostaną wstrzymane na wykonywaniu operacji READ w oczekiwaniu na krotkę, która nigdy się już nie pojawi. Jeśli chcemy, aby skończyły się one po wykonaniu zadania, można wprowadzić specjalną krotkę kończącą, up. postaci (0, układ).

Powinna ona być wprowadzona do przestrzeni przez ten proces INICJATOR lub proces WYKONAWCA, który stwierdzi koniec obliczeń. Odczytanie jej spowoduje wtedy zakończenie każdego innego procesu WYKONAWCA. Jednak w tym przypadku, po zakończeniu wszystkich procesów w przestrzeni pozostanie krotka kończąca. Jeśli nie znamy z góry liczby procesów WYKONAWCA, to nie sposób zdecydować, który z nich powinien tę krotkę usunąć. Może się bowiem tak zdarzyć, że po zakończeniu obliczeń, w których uczestniczyły, np. cztery procesy WYKONAWCA, okaże się, iż jest jeszcze piąty, który dopiero teraz chce się wykonać. Musi on mieć możliwość stwierdzenia, że powinien od razu się zakończyć, krotka kończąca musi więc być stale w przestrzeni.

Można próbować rozwiązać ten problem nieco inaczej wprowadzając specjalną krotkę inicjującą, którą proces INICJATOR na początku umieści w przestrzeni, a usunie po stwierdzeniu końca obliczeń. Każdy WYKONAWCA musiałby na początku sprawdzić operacją TRY_INPUT, czy krotka inicjująca jest w przestrzeni, i w razie jej braku nie rozpoczynać przetwarzania innych krotek. Niestety, takie rozwiązanie też nie jest dobre, gdyż jakiś spóźniony WYKONAWCA mógłby stwierdzić istnienie krotki inicjującej na chwilę przed zabraniem jej przez proces INICJATOR. W rezultacie przestrzeń krotek będzie pusta, ale WYKONAWCA pozostanie wstrzymany w oczekiwaniu na dane. Jest to jednak sytuacja trochę lepsza niż w przedstawionym rozwiązaniu, gdyż po zakończeniu obliczeń nie wszystkie procesy będą zawieszone lecz jedynie te, które iiaktywnily się w krótkim czasie między rozpoznaniem końca obliczeń
a usunięciem krotki inicjującej.

7.4.9 Obliczanie całki oznaczonej

Zakładamy, że proces UŻYTKOWNIK zaczyna swoje działanie od określenia granic całkowania a,b i zakresu dopuszczalnego błędu epsilon. Krotka przekazywana procesom

WYKONAWCA zawiera kolejno: dolną granicę całkowania, górną granicę całkowania, wartości funkcji w tych granicach, pole trapezu będące przybliżeniem całki oraz dopuszczalny błąd obliczeń. Wartości funkcji i pole trapezu mogą być obliczone na podstawie granic całkowania, są jednak przekazywane w ramach krotki, aby uniknąć wielokrotnego liczenia tych samych wielkości.

Krotki z wynikiem zawierają granice całkowania i obliczone pole. Proces UŻYTKOWNIK mógłby odbierać wyniki w miarę ich pojawiania się. Musiałby jednak umieć określić, czy odebrał już je wszystkie. Ponieważ z góry nie wiadomo, na ile podprzedziałów będzie podzielony przedział całkowania, proces UŻYTKOWNIK musiałby pamiętać każdy z nich. Aby tego iiniknąć, wyniki są odbierane zgodnie z kolejnością podprzedziałów. Dlatego w operacji INPUT w procesie UŻYTKOWNIK pierwszy parametr xl jest zawsze ustalony.

```
    proces UŻYTKOWNIK;
    var xl, x2,  fa,  fb,  całka,  p, epsilon: real;
    begin
                                    {wyznaczenie  wartości}
                                    {a,  b  i epsilon}
      fa := f(a);
      fb := f(b);
      OUTPUT(a,b,fa,fb,  (fa+fb)*(b-a)/2,  epsilon);
      xl := a;
      całka  := 0;
  repeat
    INPUT(xl, x2:real,  p:real);
    całka := całka + p;
    xl := x2
  until x2 = b;                {osiągnięto  górną}
                               {granicę  całkowania}
end;

proces WYKONAWCA;
var a, b, c, fa, fb, fc,  pole, epsilon:  real;
begin
  while true do  begin
    INPUT(a:real,  b:real, fa:real,  fb:real,
          pole:real, epsilon:real);
    c := (a+b)/2;              {środek przedziału}
    fc := f(c);
    polel := (fa+fc)*(c-a)/2;  {lewy trapez}
    pole2 := (fc+fb)*(c-b)/2;  {prawy trapez}
    if abs(pole  -  (polel+pole2))  >  epsilon then begin
      OUTPUT(a,  c, fa,  fc, polel,  epsilon/2);
      OUTPUT(c,  b, fc,  fb, pole2,  epsilon/2)
    end else
      OUTPUT(a,  b, pole)
  end
end;
```

8 Semafory w systemie Unix

8.1 Wprowadzenie

8.1.1 Operacje semaforowe

W systemie Unix jest dostępnych kilka mechanizmów, które mogą być używane do synchronizacji procesów. Są to niektóre funkcje systemowe, sygnały i semafory. Pewne funkcje systemowe, związane z systemem wejścia-wyjścia oraz z systemem plików (np. ioctl, fcntl, lockf, flock, link, creat, open} często są stosowane do synchronizacji procesów [Wils90]. Podobnie jak sygnały, a właściwie tylko jedna z ich form dostępna dla użytkownika — programowe przerwania, są to mechanizmy niskopoziomowe i dlatego nie będziemy ich tu omawiać.

Semafory [AT&T90b, Stev90] w systemie Unix sąistotnym uogólnieniem klasycznych semaforów Dijkstry, opisanych w rozdz. 3. Uogólnienia te powodnją, że semafory w Unixie są bardzo mocnym narzędziem, korzystanie z nich wymaga jednak dużych umiejętności i doświadczenia. Początkowo wprowadzono je tylko do wersji V systemu Unix, obecnie jednak są implementowane również we wszystkich innych wersjach tego systemu1.

Dostępny w Unixie mechanizm semaforów będziemy dalej przedstawiać korzystając z abstrakcyjnej, wysokopoziomowej notacji, wprowadzonej przez nas na potrzeby tej książki. Podamy jednak również funkcje systemu Unix, realizujące ten mechanizm. Ponadto niektóre z przykładów zapiszemy w języku C, stosując te funkcje.

W odróżnieniu od definicji Dijkstry, przypisanie wartości semaforowi w systemie Unix jest możliwe w dowolnej chwili.

Oprócz operacji P i V, na semaforze w Unixie można również wykonywać operację przechodzenia pod opuszczonym semaforem. Tę operację będziemy oznaczać przez Z (od zero). Wykonanie operacji Z(S) na semaforze S polega na wykonaniu instrukcji

- wstrzymaj działanie procesu, aż S będzie równe zeru.

Jeżeli proces będzie chciał wykonać operację Z, a semafor będzie opuszczony (S= 0), to proces natychmiast wykona tę operację i będzie mógł kontynuować obliczenia. Jeżeli semafor będzie podniesiony (S> 0), to proces zostanie wstrzymany aż do chwili, gdy wartość semafora zmniejszy się do zera. Dopiero wtedy proces zakończy wykonywanie operacji Z. W obu przypadkach wykonanie operacji Z nie zmienia wartości semafora. Jeżeli na zerową wartość semafora czeka wiele procesów, to wszystkie będą wznowione. Operacja Z odpowiada czekaniu przed podniesionym semaforem aż do chwili, gdy będzie on opuszczony i przechodzeniu pod opuszczonym semaforem bez zmieniania jego stanu.

Operacje P i V w Unixie umożliwiają zmianę wartości semafora nie tylko o 1, ale o dowolną dodatnią liczbę. Operacja V(S,n) oznacza zwiększenie semafora S o n, natomiast P(S,n) zmniejszenie, jeżeli to możliwe, semafora S o n.

Dalej będziemy używać tych samych nazw operacji, niezależnie od tego, czy dotyczą one zmiany semafora o 1, czy o dowolną liczbę naturalną. W pierwszym przypadku będą to operacje jednoparametrowe, w drugim natomiast dwuparametrowe.

Operacje P i Z występują również w postaci nieblokującej. Jeżeli proces nie może natychmiast wykonać, żądanej operacji, to rezygnuje z jej wykonania. Takie operacje

występują w postaci funkcji, których wartość zależy od sposobu wykonania operacji (true, gdy operacja została wykonana, false w przeciwnym razie). Funkcje te będziemy oznaczać przez nP i nZ. Można je zdefiniować następująco:

```
function nP(S: semaphore; n:  integer): boolean;
begin
   if S >= n then begin
     S := S -  n;
     nP := true
   end
   else nP  :=  false
end;

function nZ(S: semaphore): boolean;
begin
  nZ := S = 0
end;
```

Oczywiście, podobnie jak operacje P i V, również nP 5 nV są wykonywane w sposób niepodzielny.

8.1.2 Jednoczesne operacje semaforowe

W systemie Unix można wykonywać jednocześnie operacje na wielu semaforach, przy czym każda z nich może być inna. Sposób wykonania operacji jednoczesnej na wielu semaforach zależy od rodzaju operacji składowych. Jeżeli są nimi tylko P, V i Z (żadna nie jest operacją nieblokującą), to wykonanie operacji jednoczesnej zakończy się dopiero wtedy, gdy będzie można wykonać wszystkie operacje składowe. Inaczej mówiąc, jeżeli chociaż jedna z operacji składowych powoduje wstrzymanie procesu, to wykonanie operacji jednoczesnej zawiesza się aż do chwili wznowienia procesu. Jeżeli chociaż jedną z operacji składowych jest nP lub nZ (operacja nieblokująca), to cała operacja jednoczesna jest również nieblokująca. Jej wykonanie zależy od tego, czy wszystkie operacje składowe mogą być wykonane natychmiast. Jeżeli nie, to nie jest wykonywana żadna z nich. Nieblokujące operacje jednoczesne też są funkcjami o wartościach logicznych, mówiących o tym, czy operacja została wykonana (wartość true), czy nie (wartość false).

Operacje jednoczesne będziemy zapisywać w postaci ujętego w nawiasy kwadratowe ciągu operacji składowych oddzielonych przecinkami. Na przykład operacja [V(S1), P(S2,3), Z(S3)] wykona się dopiero wtedy, gdy jednocześnie wartość semafora S2 będzie większa od 2, a wartością semafora S3 będzie 0. Dopiero wtedy jednocześnie wartość S1 zwiększy się o 1, a wartość S2 zmniejszy się o 3. Natomiast operacja [nP(Sl), Z(S2), V(S3,2)] wykona się natychmiast. Jeżeli wartość semafora S1 będzie większa od 0 a wartość semafora S2 równa 0, to po wykonaniu operacji jednoczesnej wartość S1 zmniejszy się o 1, wartość S3 zwiększy się o 2, a wartością całej operacji będzie true. Jeżeli natomiast wartością S1 będzie 0 lub wartość S2 będzie większa od 0, to wartości semaforów nie zmienią się, a wartością całej operacji będzie false.

8.1.3 Funkcje na semaforach

Jest dostępnych wiele różnych funkcji określonych na semaforach. Podajemy tu, w naszej abstrakcyjnej notacji, najważniejsze z nich (wartościami wszystkich są nieujemne liczby całkowite).

• wart(S) — wartość semafora S,
• czekP(S) — liczba procesów oczekujących pod semaforem S na wykonanie operacji P,
• czekZ(S) — liczba procesów oczekujących pod semaforem S na wykonanie operacji Z.

8.1.4 Realizacja

Wartości semaforów w systemie Unix są ograniczone z góry. Jeżeli semafor jest realizowany jako zmienna typu short int, to może on przyjmować tylko wartości mniejsze od 32768.

Aby używać semaforów w programie w języku C, trzeba do programu włączyć za pomocą dyrektywy #include pliki sys/types.h, sys/ipc.h i sys/sem.h. Pliki te zawierają deklaracje typów, stałych, struktur danych i nagłówków funkcji wykorzystywanych przez operacje na semaforach.

Zbiory semaforów

Semafory używane w programie współbieżnym można pogrupować w zbiory, jednoznacznie identyfikowane przez nieujemne liczby całkowite. W każdym zbiorze są one ponumerowane kolejnymi liczbami całkowitymi, poczynając od 0. Operacje jednoczesne można wykonywać na semaforach należących tylko do tego samego zbioru. Możliwe jest wielokrotne użycie tego samego semafora w jednej operacji jednoczesnej, ale wynik takiej operacji będzie zależał od kolejności operacji składowych (por. zadanie 8.3.1).

Zanim proces zacznie korzystać ze zbioru semaforów, musi uzyskać do niego dostęp, co następuje po wykonaniu funkcji

```
int  semget  (long key, int  nsems, int semflg);
```

Parametrami funkcji semget są: key — numer zbioru semaforów w systemie (różne procesy,ktore chcą korzystać z tego samego zbioru, muszą użyć tego samego numeru), nsems — liczba semaforów w zbiorze, semflg — znaczniki określające sposób wykonania funkcji oraz prawa dostępu do zbioru semaforów. Najczęściej używanym znacznikiem jest IPC_CREAT. Oznacza on utworzenie zbioru semaforów, jeśli jeszcze nie istnieje, lub uzyskanie dostępu do już istniejącego zbioru. Prawa dostępu, określone liczbą aktualną, mówią, które procesy mają prawo odczytywać lub zmieniać wartości semaforów w zbiorze. Przykładowo 0666 oznacza możliwość wykonywania wszystkich operacji przez wszystkie procesy. Znaczniki i prawa dostępu mogą być sumowane za pomocą operatora alternatywy bitowej |.

Jeżeli zbiór semaforów o mimerze key w systemie jeszcze nie istnieje, to funkcja semget tworzy go. Jeżeli zbiór utworzono lub gdy proces uzyskał dostęp do istniejącego już zbioru, to wartością funkcji jest identyfikator zbioru, w przeciwnym razie liczba —1.
Na przykład instrukcja

```
seml  = semget  (1,  3, IPC_CREAT  |  0666);
```

wykonana po raz pierwszy spowoduje utworzenie zbioru 3 semaforów o numerze 1 i przypisanie zmiennej seml identyfikatora tego zbioru. Jeżeli zaś po wykonaniu tej instrukcji zostałaby wykonana instrukcja

```
sem2 = semget  (1,  3, IPC_CREAT  |  0666);
```

to zmienna sem2 wskazywałaby ten sam zbiór semaforów, co zmienna seml.

Funkcje na semaforach

Nadanie semaforowi wartości, odczytanie wartości semafora, odczytanie liczby procesów czekających na jego podniesienie i liczby procesów czekających na jego opuszczenie uzyskuje się w wyniku wywołania funkcji

```
int semctl (int semid, int semnum, int cmd, int val);
```

Znaczenie parametrów jest następujące: semid jest identyfikatorem zbioru semaforów, semnum — numerem semafora w zbiorze, cmd — kodem operacji wykonywanej na tym semaforze, val — parametrem operacji (uwaga: w przypadku operacji, których tu nie opisujemy, znaczenie tego parametru jest bardziej skomplikowane). Parametr cmd może przyjmować następujące wartości: SETVAL — nadanie wartości semaforowi, GETVAL — odczytanie wartości semafora, GETNCNT — odczytanie liczby procesów czekających na podniesienie semafora, GETZCNT — odczytanie liczby procesów czekających na opuszczenie semafora. Parametr val ma znaczenie tylko podczas nadawania wartości semaforowi i zawiera tę wartość. W przypadku wykonywania operacji GETVAL, GETNCNT i GETZCNT wartością funkcji semctl są odpowiednie liczby (nieujemne). Jeżeli operacja nie może być wykonana, to wartością funkcji jest —1. Tak jest na przykład wtedy, gdy usiłujemy nadać semaforowi wartość ujemną lub większą od górnego ograniczenia.

Operacje semaforowe

Operacje P, V, Z, nP i nZ wykonuje się wywołując funkcję:

```
int  semop (int semid,  struct  sembuf *sops,  unsigned nsops);
```

Jej parametrami są: semid — identyfikator zbioru semaforów; sops — wskaźnik do tablicy struktur o następującej deklaracji:

```
struct sembuf  {
  int  sem_num;
  int  sem_op;
  int  sem_flg;
};
```

Parametr nsops to liczba elementów tablicy, czyli liczba semaforów, na których ma być wykonana operacja. Znaczenie pól struktury sembuf jest następujące: sem_num — numer semafora, na którym ma być wykonana operacja, sem_op — kod operacji, sem_flg — znacznik informujący o sposobie wykonania operacji. Operacje są kodowane w następujący sposób: jeśli wartość pola sem_op jest dodatnia, to ma wykonać się operacja V(sem_num, sem_op), jeśli wartość pola sem_op jest xijemna, to ma wykonać się operacja P(sem_num, -sem_op), natomiast jeśli wartość pola sem_op wynosi 0, to ma wykonać się operacja Z(sem_num). Znacznik sem_flg może być równy stałej IPC_NOWAIT i wtedy operacja jest nieblokująca albo może być równy 0 i wtedy operacja jest blokująca.

Wartością funkcji semop jest 0, gdy operacja została wykonana, —1 w przeciwnym razie.

Powody niewykonania operacji semaforowej mogą być różnorakie — dla nas interesujące są dwa. Po pierwsze, przynajmniej jedna z operacji składowych operacji wektorowej jest nieblokująca, a operacji wektorowej nie można wykonać natychmiast. Po

drugie, zwiększenie wartości semafora (operacja V) spowodowałoby przekroczenie górnego ograniczenia.

8.1.5 Ograniczenia

Mechanizm semaforów w Unixie jest bardzo silny w porównaniu z klasycznymi semaforami. Nie jest on jednak pozbawiony pewnych ograniczeń i wad. Najpoważniejszą z nich jest brak żywotności implementacji niektórych operacji jednoczesnych (np. jednoczesnego wykonania dwóch operacji P por. 8.2.4). Jeżeli chcemy uniknąć tej wady, to musimy operacjejednoczesne zaimplementować sami, na przykład tak jak w zadaniu 3.3.4. Możemy przy tym skorzystać z pewnych ułatwień, które dają semafory unixowe, jak na przykład odczytywanie wartości semafora.

Używając semaforów unixowych musimy stosować różne funkcje do ich tworzenia i do nadawania im wartości początkowych. Ponieważ wykonywanie operacje semaforowych ma sens dopiero po zainicjowaniu semaforów, więc proces, który utworzył semafory powinien również je zainicjować, zanim inne procesy zaczną z nich korzystać. Jest kilka metod radzenia sobie z tą trudnością. Jeżeli program współbieżny tworzą procesy pokrewne (powstałe w wyniku operacji fork), to procesy korzystające z semaforów powinny być utworzone dopiero po nadaniu semaforom wartości początkowych. JeżeU nie mamy do czynienia z procesami pokrewnymi, to możemy przygotować specjalny proces, tylko inicjujący semafory, wykonywany przed rozpoczęciem innych procesów. Jeżeli nie możemy wymusić kolejności uruchamiania procesów, to możemy proces inicjujący semafory zsynchronizować za pomocą sygnałów z procesami korzystającymi z semaforów.

Ponadto wadą semaforów unixowych jest dopuszczenie operacji jednoczesnych wykonywanych na tym samym semaforze, przy nieokreślonej semantyce takich operacji. Może to prowadzić do zaskakujących wyników (por. 8.4.1).

Pewnym ograniczeniem semaforów unixowych jest brak synchronizacji typu OR. Możemy temu zaradzić implementując taki mechanizm za pomocą semaforów (por. 3.3.5), jest to jednak bardzo żmudne. Możemy również skorzystać z komunikatów jako mechanizmu synchronizacji (por. rozdz. 9). Jeżeli jednak potrzebujemy synchronizacji raz typu AND, a raz typu OR na tych samych obiektach, to nie pozostaje nam nic innego, jak samodzielne jej zaimplementowanie.

* * *

W Unixie procesy mogą komunikować się za pośrednictwem zmiennych globalnych, znajdujących się w pamięci dzielonej. Wymaga to używania funkcji systemowych, których tu nie opisujemy. Tego typu komunikacji musi jednak towarzyszyć synchronizacja, np. za pomocą semaforów. Jeżeli więc do komunikacji wystarczy tylko kilka całkowitych lub logicznych zmiennych prostych, to zamiast pamięci dzielonej jako mechanizmu komunikacji można użyć semaforów. Jest to możliwe dzięki niestandardowym operacjom semaforowym w Unixie.

W przykładach i zadaniach zamieszczonych w tym rozdziale będziemy zakładać, że semafory są jedynymi obiektami, za pomocą których procesy się synchronizują i komunikują. Niektóre semafory będą więc pełnić rolę globalnych zmiennych całkowitych. Operacje V(S,m) i P(S,m) będą zatem odpowiednio zwiększać i zmniejszać zmienną S o stałą m, a funkcja wart(S) umożUwi odczytanie jej wartości. Przyjęte podejście dla wielu czytelników może być kontrowersyjne, zarówno ze względów metodologicznych, jak i efektywnościowych (czasami zaproponowane podejście jest kosztowniejsze niż korzystanie z pamięci dzielonej). Jednak po

zapoznaniu się z mechanizmem pamięci dzielonej w Unixie (którego tu, ze względu na brak miejsca, nie opisujemy), można łatwo zmodyfikować przedstawione rozwiązania.

8.2 Przykłady

8.2.1 Wzajemne wykluczanie

Rozwiązanie problemu wzajemnego wykluczania jest takie samo, jak za pomocą standardowych semaforów (por. 3.2.1). Podamy tutaj jedynie przykładowy kod funkcji w języku C, które mogą być użyte do wzajemnego wykluczania.

Kod składa się z czterech funkcji, zgrupowanych w pliku semafor.c, który może być oddzielnie kompilowany i łączony z innymi plikami wynikowymi. Plik zawiera następujące funkcje: deklaracja, inicjacja, P oraz V. Działają one na strukturach danych, lokalnych dla tego pliku (klasa pamięci static).

Każdy proces, który chce korzystać z semafora, musi podczas inicjacji wywołać funkcję deklaracja. W funkcji tej proces uzyskuje identyfikator zbioru semaforów. Jeżeli zbiór jeszcze nie istnieje, to jest tworzony. Jeden z procesów musi wywołać funkcję inicjacjaz parametrem 1. Jej wywołanie musi nastąpić po wywołaniu funkcji deklaracja przez ten proces oraz przed wywołaniami funkcji P i V przez jakikolwiek proces. 0 tym, który z procesów powinien inicjować semafor pisaliśmy w p. 8.1.5.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

static int sera;
static struct sembuf buf[l];

void deklaracja (void)
{
  sem = semget  (1,  1,  IPC_CREAT  | 0666);
  buf[0].sem_num =  0;
  buf[0].sem_flg =  0;
}

void inicjacja  (int  i)
{
  semctl (sem,  0, SETVAL,  i);
}

void P (void)
{
  buf[0].sem_op = -1;
  semop (sem, buf,  1);
}

void V (void)
{
  buf[0].sem_op = 1;
  semop (sem, buf,  1);
}
```

Przedstawiony kod jest mało elastyczny. Mamy tu do czynienia z jednym tylko semaforem. Rozszerzenie dla wielu semaforów jest jednak bardzo proste.

8.2.2 Producenci i konsumenci

Idea przedstawionego rozwiązania jest taka sama jak rozwiązania za pomocą klasycznych semaforów. Różnice polegają na zapisaniu w odrębnych procedurach kodu służącego do synchronizacji. Poza tym zmienne J i K wskazujące element, do którego producent ma wstawiać porcję, i element, z którego konsument ma ją pobierać, są zaimplementowane jako semafory. W przykładzie 3.2.2 były to bowiem zmienne globalne i w systemie Unix musiałyby być umieszczone we wspólnej pamięci. Oczywiście bufor musi znajdować się we wspólnej pamięci, ale same operacje wstawiania do bufora i pobierania z niego nie są dla nas interesujące, zajimijemy się tu bowiem tylko synchronizacją procesów.

```
const N = ?;
      LP = ?;
      LK = ?;

var bufor: array[l..N] of  porcja;
    WOLNE: semaphore :=  N;
    PEŁNE: semaphore :=  0;
    J:  semaphore := 1;
    K:  semaphore := 1;
    CHRON_J:  semaphore  :=  1;
    CHRON_K:  semaphore  :=  1;

function pocz_prod: integer;
begin
  [ P(WOLNE),  P(CHRON_J) ];
  pocz_prod := wart(J)
end;

procedure kon_prod;
begin
  if wart(J)  = N then J  := 1
                  else V(J);    {J :=  (J + 1)  mod  N}
  [ V(CHRON_J), V(PELNE) ]
end;

function pocz_kons: integer;
begin
  C P(PELNE),  P(CHRON.K) ];
  pocz_kons := wart(K)
end;
procedure kon_kons;
begin
  if  wart(K)=N then K := 1 else V(K);  {K  := (K + 1) mod N>
  [ V(CHRON_K), V(WOLNE) ]
end;

process PRODUCENT (i: l..LP);
var p:  porcja;
begin
  while true do begin
    produkuj(p);
    bufor[pocz_prod] := p;
    kon_prod
  end
end;

process KONSUMENT (i: l..LK);
var p:  porcja;
```

```
begin
  while true do begin
    p := bufor[pocz_kons];
    kon_kons;
    konsumuj(p)
  end
end;
```

Jednoczesne operacje semaforowe można tu zastąpić operacjami sekwencyjnymi. Jednak kod w języku C operacji jednoczesnej jest bardziej zwarty niż kod odpowiadających jej operacji sekwencyjnych.

8.2.3 Czytelnicy i pisarze

Rozwiązanie z możliwością zagłodzenia pisarzy

Silny mechanizm semaforów w Unixie umożliwia rozwiązanie tego problemu za pomocą tylko jednego semafora. Jego wartość będzie określała liczbę wolnych miejsc w czytelni. Każdy czytelnik potrzebuje tylko jednego miejsca, pisarz natomiast musi zająć całą czytelnię, a więc potrzebuje wszystkich miejsc. Początkowa wartość semafora będzie większa niż liczba czytelników. Dzięki temu każdy czytelnik będzie mógł wejść do czytelni, jeśli nie będzie tam pisarza, pisarz natomiast będzie mógł wejść, jeśli nie będzie tam nikogo. Zatem ciągły napływ czytelników, gdy już jacyś czytelnicy są w czytelni, spowoduje zagłodzenie pisarzy.

```
const LC = ?;
      LP = ?;
      N = LC;
var  M:  semaphore  :=  N;  {liczba wolnych miejsc w  czytelni}

process CZYTELNIK  (i:  l..LC);
begin
 while true do begin
    wlasne_sprawy;
    P(M);               {początek  czytania}
    czytanie;
    V(M)                {koniec czytania}
 end
end;

process PISARZ (i:  l..LP);
begin
 while true do begin
    wlasne_sprawy;
    P(M,N);             {początek  pisania}
    pisanie;
    V(M,N)              {koniec pisania}
 end
end ;
```

Rozwiązanie z możliwością zagłodzenia czytelników

Jeżeli priorytet mają pisarze, to czytelnik, który chce wejść do czytelni i musi czekać, aż nie będzie czekających pisarzy. Trzeba ich zatem liczyć. Do zapamiętywania tej liczby użyjemy semafora. Jego wartość jest zwiększana o 1 przez pisarza, który chce wejść do czytelni, oraz zmniejszana o 1 przez pisarza w chwili wchodzenia do czytelni. Dla czytelnika

dodatkowym warunkiem wejścia do czytelni jest zerowa wartość tego semafora. W ten sposób pisarz, który chce wejść do czylelni przerywa wchodzenie czytelników. Jednak w tym przypadku ciągły naplyw pisarzy, gdy już jakiś pisarz jest w czytelni, spowoduje zagłodzenie czytelników.

```
    const LC = ?;
          LP = ?;
          N = LC;
    var  M:  semaphore  :=  N;  {liczba wolnych miejsc w  czytelni}
         W:  semaphore  :=  0;  {liczba czekających pisarzy}
   process CZYTELNIK  (i:  l..LC);
   begin
     while true  do  begin
       wlasne_sprawy;
       [ P(M), Z(W) ];     {początek czytania>
       czytanie;
       V(M)                {koniec czytania}
     end
   end;

   process PISARZ  (i:  l..LP);
   begin
     while true  do  begin
       wlasne_sprawy;
       V(W);               {początek pisania}
       [ P(M,N), P(W)  ];
       pisanie;
       V(M,N)              {koniec pisania}
     end
   end
```

Podajemy tutaj również kod w języku C funkcji deklaracji i inicjacji semaforów oraz rozpoczynania i kończenia czytania i pisania dla tej wersji problemu czytelników i pisarzy. Semafor M ma numer 0, a semafor W numer 1. Lokalna funkcja przygotuj służy do wypełnienia elementu tablicy buf przed wykonaniem operacji semaforowych.

```
   #include  <sys/types.h>
   #include  <sys/ipc.h>
   #include  <sys/sem.h>
   static int N  =  10000;
   static int M  = 0;     /*  semafor o numerze 0 */
   static int W  =  1;     /*  semafor o numerze 1 */
   static int sem;
   static struct sembuf buf[2];

   void deklaracja  (void)
   {
 /*  utworzenie dwuelementowego zbioru semaforów */
     sem = semget  (1,  2, IPC_CREAT |  0666);
   }
   void inicjacja (void)
   {
     semctl (sem, M, SETVAL, N);     /* M:  semaphore := N */
     semctl (sem, W, SETVAL, 0);     /* W:  semaphore := 0 */
   }

   static void przygotuj (int i,  int sem_num, int sem_op)
   {
     buf[i].sem_num = sem_num;
```

```
  buf[i].sem_op  = sem_op;
  buf[i].sem_flg = 0;
}

void pocz_czytania (void)
{
  przygotuj (0, M, -1);     /*  przygotowanie do P(M) */
  przygotuj (1, W, 0);     /*  przygotowanie do Z(W) */
  semop (sem,  buf, 2);     /*  jednoczesne wykonanie */
}

void koniec_czytania (void)
{
  przygotuj (0, M, 1);     /*  przygotowanie do V(M,1) */
  semop (sem,  buf, 1);     /*  wykonanie */
}

void pocz_pisania (void)
{
  przygotuj (0, W, 1);     /*  przygotowanie do V(W) */
  semop (sem,  buf, 1);     /*  wykonanie */
  przygotuj (0, M, -N);     /*  przygotowanie do P(M,N) */
  przygotuj (1, W, -1);     /*  przygotowanie do P(W) */
  semop (sem,  buf, 2);     /*  jednoczesne wykonanie */
}

void koniec_pisania (void)
{
  przygotuj (0, M, N);     /*  przygotowanie do V(M,N) */
  semop (sem,  buf, 1);     /*  wykonanie */
}
```

Poprawne rozwiązanie problemu czytelników i pisarzy jest tematem zadania 8.3.3.

8.2.4 Pięciu filozofów

Rozwiązanie z możliwością zagłodzenia

Podobnie jak w rozwiązaniu z zastosowaniem standardowych semaforów, każdy widelec będzie reprezentowany przez semafor WIDELEC [i]. Semafor podniesiony będzie oznaczał widelec leżący na stole. Podniesienie widelców będzie zatem jednoczesną operacją P na odpowiednich semaforach, a odłożenie — operacją V na semaforach, która może być wykonywana zarówno jednocześnie, jak i sekwencyjnie. Zauważmy, że gdyby operacje jednoczesne na semaforach unixowych miały cechę żywotności, to następujące rozwiązanie nie powodowałoby zagłodzenia.

```
var  WIDELEC: array[0..4] of semaphore := (1,1,1,1,1);

process FILOZOF  (i:  0..4);
begin
  while true do begin
     myślenie;
     [  P(WIDELEC[i]), P(WIDELEC[(i+l) mod 5]) ];
     jedzenie;
     C  V(WIDELEC[i]), V(WIDELEC[(i+l) mod 5]) ]
  end
end;
```

Rozwiązanie poprawne.

Rozwiązanie poprawne jest dokładnie takie samo, jak w przypadku semaforów Dijkstry (por. 3.2.4). Wykonywanie kolejnych operacji P musi odbywać się sekwencyjnie. Jak pamiętamy z rozdz. 2, jednoczesne podnoszenie widelców prowadzi do zagłodzenia filozofów. Warto zauważyć, że gdyby wszystkie operacje P były wykonywane jednocześnie, tzn.

```
[P(LOKAJ), P(WIDELEC[i]), P(WIDELEC[(i+l) mod 5])],
```

to wartość semafora LOKAJ mogłaby zmniejszyć się co najwyżej do 2, gdyż tylko dwóch filozofów może jednocześnie jeść. Jednak nawet rozwiązanie z następującą sekwencją operacji:

```
 P(LOKAJ);
[ P(WIDELEC[i]), P(WIDELEC[(i+l) mod 5]) ],
```

nie chroni przed efektem spisku (por. 2.6.3).

8.2.5 Implementacja monitora ograniczonego

Jak wynika z poprzednich przykładów, mechanizm semaforów w Unixie jest bardzo mocny. Rozwiązania wielu problemów można zapisać znacznie prościej i krócej niż za pomocą standardowych semaforów. Semafory w Unixie są jednak nadal obarczone wszystkimi wadami semaforów standardowych. Często jest znacznie wygodniej rozwiązać problem używając mechanizmu monitorów, a potem przetłumaczyć rozwiązanie na operacje semaforowe. Takie tłumaczenie jest całkowicie mechaniczne, jeżeli potrafimy zaimplementować monitor za pomocą semaforów. Oczywiście można to zrobić używając standardowych semaforów, ale wykorzystanie możliwości semaforów dostępnych w Unixie ułatwia rozwiązanie problemn. Przedstawimy implementację monitora z pojedynczą zmienną typu condition i operacją signal wykonywaną jedynie bezpośrednio przed wyjściem procesu z monitora. Implementacja monitora bez tego ostatniego ograniczenia jest treściąjednego z zadań. Warto porównać to rozwiązanie z rozwiązaniem za pomocą klasycznych semaforów, przedstawionym w [BenA89].

Treść każdej procedury eksportowanej przez monitor musi być poprzedzona wywołaniem procedury wejście i musi ją kończyć albo wywołanie procedury signal (gdy ostatnią instrukcją procedury eksportowanej jest signal), albo wywołanie procedury wyjście (gdy tak nie jest).

```
var  W:  semaphore :=  1;  {do wzajemnego wykluczania}
     L:  semaphore :=  0;  {do liczenia procesów czekających na warunek}
     C:  semaphore :=  0;  {do oczekiwania na warunek}

procedure wejście;
begin
  P(W)
end;

procedure wait;
begin
  [  V(W),  V(L) ];
  P(C)
end;
```

```
   procedure signal;       {signal i wyjście}
   begin
     if not  [ nP(L), V(C)  ]  then V(W)
   end;

   function  empty: boolean;
   begin
     empty := wart(L)  = 0
   end;
   procedure wyjście;     {wyjście bez signal}
   begin
     V(W)
   end;
```

Semafor L służy tylko do liczenia procesów czekających pod semaforem C na spełnienie warunku, więc na pierwszy rznt oka wydaje się, że jest on zbędny. Można by przecież w procedurze signal zastosować instrukcję:

```
   if czekP(C) > 0 then V(C) else V(W)
```

Mogłoby się jednak zdarzyć, że proces wykonujący procedurę wait nie rozpoczął jeszcze operacji P(C), a już inny wykonujący procedurę signal (która mogła przecież rozpocząć się po rozpoczęciu operacji wait) stwierdził, że żaden proces nie czeka pod semaforem C. Oznaczałoby to, że operacja V(C) nie będzie wykonana i proces, który dokończy wait wykonując P(C), pozostanie wstrzymany. Sprawdzanie warunku w procedurze signal jest bezpieczne, gdyż odbywa się po przejściu pod semaforem W, a więc w monitorze.

Warto porównać powyższe rozwiązanie z rozwiązaniem przedstawionym w [Tane92], w którym brak jest liczenia procesów wstrzymanych w monitorze.

W rezultacie procedura wait składa się tylko z instrukcji V(W) ; P(C), a signal — z bezwarunkowo wykonywanej instrukcji V(C). Jest zatem możliwe równoczesne wykonywanie procedur monitora przez wiele procesów, a także całkowite zablokowanie monitora.

A oto kod podanych procedur w języku C. Funkcja przygotuj ma tu o jeden parametr więcej niż w przykładzie 8.2.3, gdyż potrzebne są operacje nieblokujące.

```
   #include  <sys/types.h>
   #include  <sys/ipc.h>
   #include  <sys/sem.h>

   static  int  W = 0;              /* semafor o numerze 0 */
   static  int  L = 1;              /* semafor o numerze 1 */
   static  int  C = 2;              /* semafor o numerze 2 */
   static  int  sem;
   static  struct sembuf buf[3];
   void deklaracja  (void)
   {
  /* utworzenie trzyelementowego zbioru semaforów */
     sem = semget(l, 3,  IPC_CREAT | 0666);
   }
void inicjacja (void)
{
  semctl (sem, W,  SETVAL,  1);  /*  W:  semaphore  :=  1  */
  semctl (sem, L,  SETVAL,  0);  /*  L:  semaphore  :=  0  */
  semctl (sem, C,  SETVAL,  0);  /*  C:  semaphore  :=  0  */
}

static void przygotuj  (int i,  int sem_num, int sem_op,  int  sem_flg)
```

```
{
  buf[i].sem_num = sem_num;
  buf[i].sem_op  = sem_op;
  buf[i].sem_flg = sem_flg;
}

void wejście (void)
{
  przygotuj(0, W,  -1,  0);     /* przygotowanie  do  P(W)  */
  semop(sem, buf,  1);          /* wykonanie */
}

void wait (void)
{
  przygotuj(0, W,  1,  0);       /*  przygotowanie do V(W)  */
  przygotuj(l, L,  1,  0);       /*  przygotowanie do V(L)  */
  semop(sem, buf,  2);           /*  jednoczesne wykonanie  */
  przygotuj(0, C,  -1,  0);      /*  przygotowanie do P(C)  */
  semop(sem, buf,  1);           /*  wykonanie */
}

void signal (void)
{
  przygotuj(0, L,  -1,  IPC_NOWAIT);   /*  przygotowanie do nP(L) */
  przygotuj(l, C,  1,  0);              /*  przygotowanie do V(C)  */
  if (!semop(sem,  buf,  2))            /*jednoczesne wykonanie  */
  {
     przygotuj(0,  W,  1, 0);           /*  przygotowanie do V(W)  */
     semop(sem, buf,  1);               /*  wykonanie */
  }
}
   int  empty (void)
   {
                                  /* ostatni parametr nie ma */
     return                       /* znaczenia  */
        !semctl(sem,L,GETVAL,0);  /* wart(L) */
   }

   void wyjście (void)
   {
     przygotuj(0, W, 1,  0);      /* przygotowanie do V(W) */
     semop(sem, buf, 1);          /* wykonanie  */
   }
```

8.3 Zadania

Rozwiązania poniższych zadań podajemy nżywając abstrakcyjnej notacji. Czytelnika zachęcamy do zapisania ich również w języku C, z wykorzystaniem funkcji systemu Unix.

8.3.1 Implementacja semafora binarnego

Zapisz operacje PB \ VB na semaforze binarnym za pomocą operacji semaforowych w systemie Unix.

8.3.2 Implementacja semafora typu OR

Zaimplementuj semafor typn OR za pomocą semaforów w systemie Unix.

8.3.3 Czytelnicy i pisarze — rozwiązanie poprawne

Rozwiąż problem czytelników i pisarzy w wersji bez zagłodzenia za pomocą semaforów w systemie Unix.

8.3.4 Implementacja monitora ogólnego

Za pomocą semaforów w systemie Unix zaimplementuj monitor, w którym dopuszcza się wykonanie operacji signalw dowolnym miejscu procedury monitora.

8.3.5 Zasoby dwóch typów

Rozwiąż zadanie 4.3.10 za pomocą semaforów w systemie Unix.

8.4 Rozwiązania

8.4.1 Implementacja semafora binarnego

Semafor binarny jest podobny do semafora dwustronnie ograniczonego o wartości maksymalnej 1. Różnica, zgodnie z przyjętą w tej książce definicją semafora binarnego, polega na przerwaniu przez procedurę błąd procesu, który usiłuje podnieść już podniesiony semafor.

```
var S:  semaphore  := 1;   {początkowo podniesiony}
    T:  semaphore  := 0;

procedure PB;
begin
   P(S);
   V(T)
end;

procedure VB;
begin
   if  nP(T) then .V(S)
   else  błąd
end;
```

Procedura VB może być też zapisana prościej:

```
procedure VB;
begin
   if  not [ V(S), nP(T) ] then błąd
```

```
end;
```

Alternatywna definicja semafora binarnego dopuszcza, aby podniesienie już podniesionego semafora nie bylo błędem. Jednak proces wykonujący operację VB powinien wiedzieć, czy operacja została wykonana. Można do tego użyć wartość operacji nieblokującej w procedurze VB:

```
function VB:  boolean;
begin
   VB  := C V(S), nP(T) ]
end;
```

Jeżeli uważnie przeanalizujemy powyższe rozwiązanie, to wydaje się, że wystarczy tylko jeden semafor:

```
var S:  semaphore := 0;

procedure PB;
begin
  P(S)
end;

function VB:  boolean;
begin
  VB  := [ nZ(S), V(S) ]
end;
```

W tym rozwiazaniu usiłuje się podnieść semafor S tylko wtedy, gdy jest opuszczony. Jednak w dokumentacji systemu Unix semantyka równoczesnych operacji na tym samym semaforze nie jest określona. Poprawność rozwiązania zależy od kolejności poszczególnych operacji składowych w operacji jednoczesnej. Testy wykazują, że operacja VB zaimplementowana w następujący sposób:

```
static  int  sem;
static  struct sembuf buf[2];

int VB  (void)
{
  buf[0].sem_num = 0;   /* równoważne   [  nZ(S), V(S) ]  */
  buf[0].sem_op   = 0;
  buf[0].sem_flg = IPC_NOWAIT;
  buf[l].sem_num = 0;
  buf[l].sem_op   = 1;
  buf[l].sem_flg = 0;
  return semop  (sem, buf, 2);
}
```

zawsze nadaje semaforowi wartość 1, a jej wynikiem jest 0. Natomiast ta sama operacja zaimplementowana następująco:

```
static  int  sem;
static  struct sembuf buf[2];

int VB  (void)
{
  buf[0].sem_num = 0;   /* równoważne   [  V(S), nZ(S) ]  */
  buf[0].sem_op   = 1;
```

```
      buf[0].sem_flg  =  0;
      buf[l].sem_num  =  0;
      buf[l].sem_op   =  0;
      buf[l].sem_flg  =  IPC_NOWAIT;
      return semop  (sem,  buf, 2);
   }
```

zawsze nadaje semaforowi wartość 0, a jej wynikiem jest —1.

Podobny problem powstaje, gdy chcemy zaimplementować semafor dwustronnie ograniczony o górnej wartości 1, używając, tylko jednego semafora. Poprawnie zadziała taka oto implementacja:

```
   var S:  semaphore  := 0;

   procedure PD;
   begin
      P(S)
   end;

   procedure VD;
   begin
      [ Z(S), V(S) ]
   end;
```

Natomiast operacja VD zaiplementowana w następujący sposób:

```
   procedure VD;
   begin
      C v(s), z(s) ]
   end;
```

powoduje nieskończone oczekiwanie. (Jest to ciekawy przykład blokady, w której bierze udział tylko jeden proces.)

8.4.2 Implementacja semafora typu OR

Rozwiązanie jest w zasadzie takie samo, jak w rozdz. 3 (por. 3.4.5). Różnice polegają na zastąpieniu zmiennych globalnych semaforami oraz na wyeliminowaniu niektórych zmiennych, dzięki temu, że w Unixie można odczytywać wartości semaforów i wykonywać operacje nieblokujące. W przeciwieństwie do rozwiązania z rozdz. 3, zwykła operacja P na pojedynczym semaforze pozostaje niezmieniona.

```
const M = ?;
      N = ?;
var
S: array [0..1] of semaphore := (N,M); {semafory właściwe}
LUB: semaphore := 0;   {do wstrzymywania procesów, gdy oba semafory są opuszczone}
ILE: semaphore := 0;   {do liczenia czekających pod LUB}
WYK: semaphore := 1;   {do wzajemnego wykluczania podczas wykonywania operacji
semaforowych}
WYBRAŁ: semaphore := O;{wskazuje, który semafor podniesiono w VSINGLE}
COBYŁO: array[0..1]  of semaphore := (1,1);
                       {do pamiętania poprzednich wyborów, aby uniknąć zagłodzenia}
KTÓRY: semaphore := 0; {do zrównoważenia wykorzystania semaforów, gdy oba są
podniesione}
```

```
procedure POR (var i: 0..1);
var k: integer;
begin
  P(WYK);
  k := wart(KTÓRY);
  if nP(S[k]) then
  begin                {był podniesiony S[k]}
    i := k;
    if k = 0 then V(KTORY)
    else P(KTÓRY);   {w przyszłości będziemy najpierw sprawdzać ten drugi semafor}
    V(WYK)
  end
  else
  if nP(S[l-k]) then
  begin                {był podniesiony S[l-k]}
    i := 1 - k;
    if k = 0 then V(KTORY) else P(KTÓRY);
    V(WYK)
  end
  else begin           {opuszczone oba}
    V(ILE);
    V(WYK);
    P(LUB);
        if  nP(WYBRAL) then  i  :=  1
                       else  i  :=  0;
        V(WYK)
     end
   end;

   procedure VSINGLE (i: 0..1);
   var  c,  llubO, lsO: boolean;
   begin
     P(WYK);
     c  :=  wart(COBYLO[i])  =  1;
     llubO :=  wart(ILE) =  0;
     lsO  := czekP(S[i]) =  0;
     if not llubO and not  lsO then
        if  c then P(COBYLO[i])
             else V(COBYLO[i]);
     if not llubO and (c or  lsO)
     then            {ktoś czeka na dowolny oraz ma on tym razem pierwszeństwo
                      lub nikt nie czeka na S[i]}
     begin
        if  i =  1 then V(WYBRAL[i]);
        P(ILE);
        V(LUB)
     end else  begin
        V(S[i]);
        V(WYK)
     end
   end;
```

Mogłoby się wydawać, że można tutaj zrezygnować z semafora ILE i Zastąpić warunek wart(ILE) = O warunkiem czekP(LUB) = 0. Jednak proces wykonujący procedurę POR w sytuacji, gdy oba semafory S są opuszczone, może podnieść semafor WYK, i nie rozpocząć natychmiast oczekiwania pod semaforem LUB. Jeżeli teraz wykona się procedura VSINGLE i żaden proces nie czeka pod semaforem S, to ten semafor zostanie podniesiony. Proces, który teraz zacznie oczekiwanie pod semaforem LUB będzie niepotrzebnie wstrzymywany, mimo że jeden z semaforów S jest podniesiony.

Jak widać, implementacja semafora typu OR za pomocą semaforów unixowych jest równie złożona, jak za pomocą semaforów klasycznycli. Dlatego w razie konieczności stosowania synchronizacji typu OR lepiej jest używać mechanizmu komunikatów w systemie Unix (por. 9.3.1).

8.4.3 Czytelnicy i pisarze — rozwiązanie poprawne

Aby uniknąć zagłodzenia czytelników, wprowadzimy przed czytelnią niewielki przedsionek, w którym może przebywać co najwyżej jeden oczekujący na wejście pisarz. Pisarz może wejść do przedsionka tylko wtedy, gdy ani w przedsionku, ani w czytelni nie ma innego pisarza. Pisarz czeka w przedsionku, aż ostatni czytelnik opuści czytelnię, po czym sam do niej wchodzi. Czytelnicy czekają na wejście do czytelni zarówno wtedy, gdy w środku jest pisarz, jak i wtedy, gdy pisarz jest w przedsionku. Jeżeli pisarz wyjdzie z czytelni, to czytelnicy będą wchodzić do niej tak długo, aż do przedsionka wejdzie jakiś pisarz.

Aby zaimplementować ten algorytm, musimy wprowadzić dodatkowy semafor do wzajemnego wykluczania pisarzy w przedsionku i w czytelni.

```
const  LC  = ?;
       LP  = ?;
       N  = LC;

var M:  semaphore := N;  {liczba wolnych miejsc}
    I:  semaphore := 0;  {liczba pisarzy w przedsionku}
    W:  semaphore :=  1; {do wzajemnego wykluczania pisarzy}

process  CZYTELNIK  (i:  l..LC);
begin
  while  true  do begin
    wlasne_sprawy;
     [  P(M),  Z(I) ];     {początek czytania}
    czytanie;
    V(M)                {koniec czytania}
  end
end;

process  PISARZ (i: l..LP);
begin
  while  true  do begin
    wlasne_sprawy;
     [  V(I),  P(W) ];     {wejście do przedsionka}
     [  P(M,N),  P(I) ];   {początek pisania}
    pisanie;
     [  V(M,N),  V(W) ]    {koniec pisania}
  end
end;
```

Nie jest to doskonałe rozwiązanie. Jego wadą jest to, że po wyjściu pisarza z czytelni nie muszą tam wejść wszyscy czekający czytelnicy. Może się nawet zdarzyć, że nie wejdzie żaden z nich, gdyż najpierw do przedsionka wejdzie kolejny pisarz. Jest to spowodowane jednoczesnym podnoszeniem semaforów M i W. Można wobec tego próbować opóźnić wchodzenie czekającego pisarza do przedsionka, podnosząc sekwencyjne najpierw semafor M, a potem W. Może się jednak wówczas zdarzyć, że jeśli po wyjściu z czytelni pisarz będzie zwlekał z podniesieniem semafora W, to do czytelni będą mogli wchodzić czytelnicy, którzy przybyli w tym czasie, wyprzedzając czekających pisarzy.

8.4.4 Implementacja monitora ogólnego

Implementacja monitora ogólnego wynika bezpośrednio z definicji monitora podanej w rozdz. 4. Implementacja monitora za pomocą klasycznych semaforów z użyciem zmiennych globalnych została podana np. w [BenA89].

Każda procedura monitora musi być poprzedzona wywołaniem procedury wejście, a zakończona wywołaniem wyjście lub signal_i_wyjscie, gdy operacja signal jest ostatnią instrukcją procedury monitora. Dla każdej zmiennej typu condition jest potrzebna para semaforów L i C, które służą do realizacji kolejki procesów wstrzymanych w oczekiwaniu na operację wait.
Semafory M i S służą do realizacji kolejki procesów wstrzymanych po wykonaniu operacji signal.

```
var  W: semaphore :=  1;  {do wzajemnego wykluczania}
     L: semaphore :=  0;  {do liczenia procesów czekających na warunek}
     C: semaphore :=  0;  {do zawieszania w oczekiwaniu na warunek}
     M: semaphore :=  0;  {do liczenia procesów wstrzymanych po signal}
     S: semaphore :=  0;  {do wstrzymywania po signal}

procedure wejście;
begin
   P(W)
end;

procedure wait;
begin
   if not  [ nP(M), V(S), V(L) ]  then [ V(W), V(L) ];
   P(C)
end;
procedure signal;
begin
   if  [ nP(L), V(C), V(M) ] then  P(S)
end;

function empty: boolean;
begin
   empty := wart(L) = 0
end;

procedure signal_i_wyjscie;
begin
   if  not C nP(L), V(C) ] then
   if  not [ nP(M), V(S) ] then V(W)
end;

procedure wyjście;  {gdy na końcu nie ma signal}
begin
   if  not [ nP(M), V(S) ] then V(W)
end;
```

Modyfikacja tego rozwiązania dla wielu zmiennych typu condition jest bardzo prosta. Zmienne trzeba ponumerować, a zamiast semaforów L i C wprowadzić tablice semaforów, indeksowane mimerami zmiennych. Do procedur wait, signal, empty oraz signal_i_wyjscie trzeba przekazywać numer zmiennej, na której ma być wykonana odpowiednia operacja.

8.4.5 Zasoby dwóch typów

Użyjemy następujących semaforów: WOLNE_A i WOLNE_B do pamiętania, ile jest wolnych zasobów każdego typu (ponadto WOLNE_A służy do wstrzymywania procesów grupy 1, gdy nie ma wolnych zasobów typu A); CZEKA_A i CZEKA_B do liczenia czekających procesów odpowiednio grupy 1 i 2; A_LUB_B do wstrzymywania procesów grupy 2, gdy nie ma dla nich wolnych zasobów; ZWOLNIONE_A i ZWOLNIONE_B do liczenia, ile zasobów każdego typu zwolniono, gdy czekał proces grupy 2; WYK do wzajemnego wykluczania podczas wykonywania procedur przydzielania i zwalniania zasobów. Powody wprowadzenia semaforów ZWOLNIONE_A i ZWOLNIONE_B są takie same, jak semaforów WYBRAŁ w implementacji semafora typu OR.

```
   const M = ?;
         N = ?;
   var WOLNE_A,  WOLNE_B:  semaphore  :=  (M,  N);
    CZEKA_A:  semaphore := 0;   {ile czeka na A}
    CZEKA_AB: semaphore := 0;   {ile czeka na A lub B}
    A_LUB_B:  semaphore := 0;   {do wstrzymywania czekających na A lub B}
    ZWOLNIONE_A, ZWOLNIONE_B: semaphore := (0, 0);
    WYK: semaphore := 1;        {do wzajemnego wykluczania}

procedure przydziel (grupa: 1..3; var typ: A..B; var dostał: boolean);
begin
  P(WYK);
  case grupa of
  1: if not nP(WOLNE_A) then begin
       [ V(CZEKA_A), V(WYK) ];
       P(WOLNE_A)
     end;
  2: if [ nP(WOLNE_A), V(WYK) ] then typ := A
     else
     if [ nP(WOLNE_B), V(WYK) ] then typ := B
     else begin
       C V(CZEKA_AB), V(WYK) ] ;
       P(A_LUB_B);
       if nP(ZWOLNIONE_A)  then typ := A
       else
       if nP(ZWOLNIONE_B)  then typ := B
     end;
  3: begin
       dostał := true;
       if nP(WOLNE_A) then typ  := A
       else
       if nP(WOLNE_B) then typ  := B
       else dostał := false;
       V(WYK)
     end
  end
end;

procedure zwolnienie (typ: A..B);
begin
  P(WYK);
  case typ of
  A: if not [ nP(CZEKA_A), V(WOLNE_A) ]  then
     if not [ nP(CZEKA_AB), V(A_LUB_B),  V(ZWOLNIONE_A)  ]
       then V(WOLNE_A);
  B:  if not [ nP(CZEKA_AB), V(A_LUB_B),  V(ZWOLNIONE_B)  ]
```

```
      then V(WOLNE_B)
  end;
  V(WYK)
end;
```

9 Komunikaty i kanały w systemie Unix

9.1 Wprowadzenie

9.1.1 Potoki i gniazda

W systemie Unix jest dostępnych kilka różnych mechanizmów komunikacji. Są to: pamięć dzielona, potoki, potoki nazwane, gniazda, komunikaty \ zdalne wywołanie procedury [AT&T90b, Stev90].

Pamięć dzielona, opisywana wszędzie równocześnie z mechanizmami komunikacji, umożliwia przesyłanie informacji między procesami, ale nie zapewnia przy tym żadnej synchronizacji. Użytkownik musi sam zadbać o wzajemne wykluczanie procesów żądających dostępu do wspólnej pamięci. Jako odrębny mechanizm nie nadaje się więc ona do programowania współbieżnego, ale można z niej korzystać w powiązaniu z mechanizmami synchronizacji, np. z semaforami.

Potoki (pipes), dostępne w każdej wersji systemu Unix, umożliwiają jednokierunkową, asynchroniczną komunikację między pokrewnymi procesami, utworzonymi za pomocą operacji fork. Informacja jest przesyłana w postaci sekwencyjnego strumienia bajtów, którego interpretacja należy do procesów korzystających z potoku. Odczytanie informacji powoduje usunięcie jej z potoku. Jeżeli w potoku nie ma tyle informacji, ile chce odczytać proces odbiorca, to jest on wstrzymywany. Potoki mogą być w łatwy sposób wykorzystywane na poziomie warstwy systemu operacyjnego odpowiedzialnej za komunikację z użytkownikiem (shell). Taki tryb korzystania z potoków przedstawimy w rozdz. 11. Implementacja potoków w systemach Unix i DOS jest odmienna, jednak koncepcyjnie oba mechanizmy są takie same.

Potoki nazwane (named pipes lub FIFOs] różnią się od zwykłych potoków tym, że pozwalają komunikować się dowolnym procesom (niekoniecznie pokrewnym). Potoki są mechanizmem scentralizowanym, natomiast gniazda (sockets) umożliwiają komunikację dwukierunkową w środowisku zarówno scentralizowanym, jak i rozproszonym.

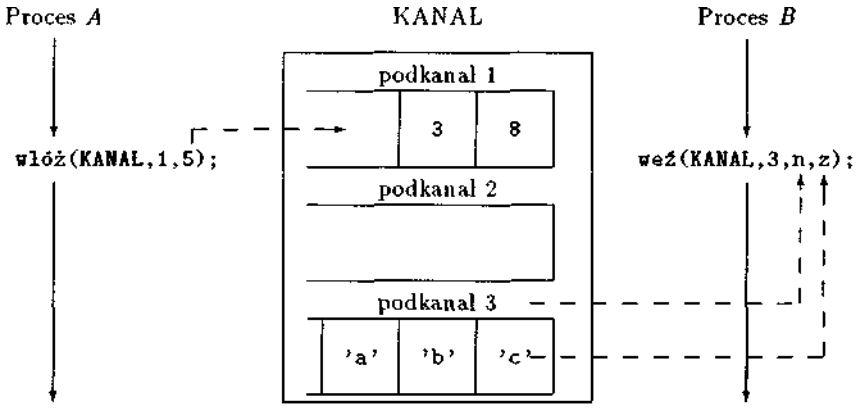
Między pojęciem potoku a pojęciem semafora można doszukać się wielu analogii. Semafor wymyślono w celu maksymalnego uproszczenia rozwiązania problemu wzajemnego wykluczania, potok — w celu uproszczenia rozwiązania problemu producenta i konsumenta. Semafor jest w rzeczywistości zmienną całkowitą, potok — plikiem. Operacja V na semaforze odpowiada operacji pisania do potoku, operacja P — operacji czytania z potoku. W przypadku semaforów zazwyczaj nie rozważa się możliwości nadmiaru przy wykonywaniu operacji V i w związku z tym mówi się, że można ją wykonać zawsze. Potok ma ograniczoną pojemność i dlatego operacja pisania do potoku może być operacją blokującą. Potok traktowany jako mechanizm synchronizacji jest więc odpowiednikiem semafora dwustronnie ograniczonego. (Podobna analogia między semaforem i buforem, w którym są umieszczane puste porcje informacji, została przedstawiona w [Brin79].)

Nie będziemy tu opisywać sposobu korzystania z potoków i gniazd. Zamiast tego omówimy ogólniejszy mechanizm komunikatów, będący strukturalnym rozszerzeniem mechanizmu potoków nazwanych. W następnym rozdziale przedstawimy zdalne wywołanie procedury (Remote Procedure Call), które w Unixie zrealizowano za pomocą mechanizmu gniazd.

Mechanizm komunikatów (messages) początkowo wprowadzono tylko do wersji V systemu Unix, ale obecnie jest dostępny również w innych wersjach tego systemu. Przedstawimy go korzystając z abstrakcyjnej, wysokopoziomowej notacji. Podamy jednak również funkcje systemu Unix, realizujące ten mechanizm. Ponadto niektóre z przykładów zapiszemy w języku C korzystając z tych funkcji.

9.1.2 Kanały i podkanały

Kanały są scentralizowanym mechanizmem komunikacji między procesami, służącym do przesyłania jednostek informacji o określonej przez użytkownika strukturze, nazywanych komunikatami (rys. 9.1). W kanale może istnieć dowolnie dużo autonomicznych podkanałów. Komunikacja przez kanały jest asynchroniczna i dwukierunkowa. Kolejność pobierania komunikatów z poszczególnych podkanałów jest taka sama, jak kolejność ich umieszczenia. Każdy proces korzystający z kanału może umieszczać komunikaty w dowolnym podkanale oraz pobierać je również z dowolnego podkanału. Pojemność kanału jest ograniczona. Jej maksymalna wartość jest ustalana podczas generowania systemu.



Rys. 9.1. Komunikacja przez kanał unixowy

Kanał jest obiektem globalnym dla każdego procesu, który z niego korzysta. W abstrakcyjnej notacji używanej w tej książce kanały będziemy deklarować na zewnątrz procesów korzystających z nich, stosując nastepujący zapis:

```
K:  channel
```

gdzie K jest identyfikatorem kanału, a channel — predefiniowanym typem. Początkowo kanał jest pusty. Podkanały są identyfikowane liczbami naturalnymi w obrębie kanału. Kanał ma pewną ustaloną pojemność. Możliwość zmiany pojemności jest ograniczona (można ją tylko zmniejszyć) i dlatego przy deklaracji kanału nie będziemy jej określać (wyjątkiem jest przykład 9.2.1).

Kolejność wznawiania procesów wstrzymanych podczas wykonywania operacji na kanałach jest nieokreślona. Specyfikacja funkcji systemu Unix nic na ten temat nie mówi, a testy praktyczne pokazują, że procesy są wznawiane w takiej samej kolejności, w jakiej były wstrzymane.

9.1.3 Operacje na kanałach

Na kanałach można wykonywać dwie operacje: umieszczanie komunikatu w kanale oraz pobieranie komunikatu z kanału. Komunikat umieszcza się w kanale wykonując operację, którą w naszej notacji będziemy zapisywać w następujący sposób:

```
wlóż  (K, podkanal, komunikat)
```

Parametry tej operacji mają następujące znaczenie: K i podkanał to identyfikatory kanału i podkanału, w których ma być umieszczony komunikat, komunikat jest przesyłanym komunikatem, którego typ określa użytkownik. Komunikat pobiera się z kanału za pomocą operacji:

```
weź (K,  podkanal, skąd, komunikat)
```

przy czym K i podkanał to identyfikatory kanału i podkanału, z których ma być pobrany komunikat. Identyfikator podkanału może występować w jednej z trzech postaci: jako liczba naturalna - jest wtedy jednoznacznym identyfikatorem podkanału, jako zero - jest wtedy identyfikatorem dowolnego podkanału, jako ujemna liczba całkowita - jest wtedy identyfikatorem podkanałów o numerach mniejszych od modułu tej liczby bądź równych jej modułowi. Jeżeli identyfikatorem podkanału jest zero, to pobiera się komunikat najdawniej umieszczony w kanale. Jeżeli jest to liczba ujemna, to pobiera się komunikat z podkanału o najmniejszym identyfikatorze mniejszym od modułu tej liczby bądź równym temu modułowi. Wynikiem operacji jest identyfikator podkanału, z którego pobrano komunikat (parametr skąd) oraz komunikat (parametr komunikat).

Obie powyższe operacje są blokujące, tzn. wstrzymują wykonywanie procesu, jeżeli komunikatu nie można umieścić w kanale albo pobrać z kanału. Proces jest wznawiany, gdy operacja może być wykonana. Istnieją również nieblokujące wersje tych operacji:

```
n_wloz (K,  podkanał, komunikat)
n_wez  (K, podkanał, skąd, komunikat)
```

Parametry tych operacji mają takie samo znaczenie jak parametry odpowiadających im operacji blokujacych. Ich wynikiem są wartości logiczne: true, jeżeli operacja została wykonana, oraz false w przeciwnym razie.

9.1.4 Realizacja

W oryginalnej terminologii systemu Unix kanał jest nazywany kolejką komunikatów, numer kanału zaś kluczem. Z każdym komunikatem jest związany typ komunikatu identyfikujący podkanał, którym jest przekazywany ten komunikat.

Chcąc używać mechanizmu komunikatów w systemie Unix trzeba do programu w języku C włączyć za pomocą dyrektywy #include pliki nagłówkowe sys/types.h, sys/ipc.hi sys/msg.h. Pliki te zawierają deklaracje typów, stałych, struktur danych i nagłówków funkcji używanych przez operacje przesyłania komunikatów.

Zanim proces zacznie korzystać z kanału musi uzyskać do niego dostęp, wykonując funkcję

```
int msgget (long key, int msgflg);
```

Parametrami są: key - numer kanału w systemie (różne procesy, które chcą korzystać z tego samego kanału, muszą użyć tego samego numeru), msgflg - znaczniki określające sposób wykonania funkcji oraz prawa dostępu do kanału. Najczęściej używanym znacznikiem jest IPC_CREAT, oznaczający utworzenie kanału (jeśli jeszcze nie istnieje) lub uzyskanie dostępu do już istniejącego kanału. Prawa dostępu, określone liczbą oktalną, mówią, które procesy mają prawo umieszczać komunikaty w kanale oraz pobierać je z kanału. Przykładowo 0666 oznacza możliwość wykonywania wszystkich operacji przez wszystkie procesy. Znaczniki i prawa dostępu mogą być sumowane za pomocą operatora alternatywy bitowej |.

Jeżeli kanał o numerze key jeszcze w systemie nie istnieje, to funkcja msgget powoduje utworzenie go. Jeżeli kanał utworzono lub gdy proces uzyskał dostęp do istniejącego już kanału, to wynikiem funkcji jest identyfikator kanału, w przeciwnym razie wynikiem jest -1.

Na przykład instrukcja

```
K1 = rasgget (1, IPC_CREAT | 0666);
```

wykonana po raz pierwszy spowoduje utworzenie kanalu o numerze 1 i przypisanie zmiennej K1 identyfikatora tego kanału. Jeżeli zaś po wykonaniu tej instrukcji wykonałaby się instrukcja

```
K2 = msgget (1, IPC_CREAT | 0666);
```

to zmienna K2 wskazywałaby ten sam kanał, co zmienna K1.

Operacje na kanałach

Operacjom włóż i n_wloz odpowiada w systemie Unix funkcja

```
int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
```

Parametr msqid jest identyfikatorem kanału, a msgp - wskaźnikiem do struktury o następującej deklaracji:

```
struct msgbuf {
    long mtype;    /* identyfikator podkanału */
                   /* treść komunikatu */
}
```

składającej się z identyfikatora mtype podkanału, do którego ma być włożony komunikat, oraz zmiennych zawierających komunikat. Tę część struktury użytkownik deklaruje w zależności od potrzeb. Dla komunikatu pustego struktura msgbuf składa się tylko z identyfikatora podkanału. Kolejne parametry funkcji msgsnd to: msgsz - wielkość przesyłanego komunikatu (czyli rozmiar zadeklarowanych zmiennych), msgflg - znacznik określający sposób wykonania operacji. Jeżeli jest ona równa stałej IPC_NOWAIT, to operacja jest nieblokująca, jeżeli natomiast jest równa 0, to operacja jest blokująca.

Wynikiem funkcji msgsnd jest 0, gdy operacja wykona się, - 1 w przeciwnym razie. Powody niewykonania mogą być różnorakie - dla nas interesujący jest jeden: operacja jest nieblokująca, a kanał całkowicie zapełniony komunikatami.

Operacjom weź i n_wez odpowiada w systemie Unix funkcja:

```
int msgrcv (int msqid, struct msgbuf *msgp, int msgsz,
            long msgtyp, int msgflg);
```

Parametr msgtyp oznacza identyfikator podkanału, z którego ma być pobrany komunikat. Znaczenie wartości tego parametru zostało podane przy opisie operacji weź. Pozostałe parametry mają takie samo znaczenie jak parametry funkcji msgsnd. W strukturze wskazywanej przez msgp znajdują się wartości funkcji msgrcv: identyfikator podkanału, z którego pobrano komunikat, oraz sam komunikat.

Wynikiem funkcji msgrcv jest 0, gdy operacja wykona się, - 1 w przeciwnym razie. Powody niewykonania mogą być różnorakie - dla nas interesujący jest jeden: operacja jest nieblokująca, a podkanał (lub podkanały), z którego ma być pobrany komunikat, jest pusty.

Pojemność kanału

Kanał ma pewną standardową pojemność, wyrażaną w bajtach, która jest parametrem systemu operacyjnego. Użytkownik może jedynie zmniejszyć pojemność kanału, natomiast zwiększyć może ją tylko administrator systemu. Suma wielkości treści komunikatów we wszystkich podkanałach nie może być większa od pojemności kanału.

Do odczytania lub nadania pojemności kanału służy funkcja

```
int msgctl (int msqid, int cmd, struct msqid_ds *bufp);
```

Parametr msqid jest identyfikatorem kanału, cmd określa rodzaj operacji (IPC_STAT - odczytanie pojemności, IPC_SET - nadanie pojemności), a bufp jest wskaźnikiem do predefiniowanej struktury, w której pole unsigned long msg_qbytes zawiera pojemność kanału. Wynikiem funkcji jest O,jeżeli jej wykonanie powiodło się, oraz -1 w przeciwnym razie.

Ponieważ funkcja msgctl może być wykorzystywana do wielu innych celów, więc struktura wskazywanaprzez bufp zawiera wiele innych pól, które muszą zawierać właściwe wartości podczas nadawania pojemności kanałowi. Najpewniejszym sposobem umieszczenia ich w tej strukturze jest wcześniejsze wywołanie funkcji msgctl z parametrem cmd równym IPC_STAT.

Podajemy teraz przykładową funkcję służącą do nadawania kanałowi K pojemności n komunikatów, które będą umieszczane w strukturze msgbuf. Określając wielkość komunikatu, odejmujemy od wielkości struktury msgbuf rozmiar pola przeznaczonego na przechowywanie identyfikatora podkanału (pole typu long).

```
pojemność (int K,  int n)
{
 struct msaid ds bnfn:
  if (msgctl  (K,  IPC_STAT, &bufp)  == 0) {
    bufp.msg_qbytes = n *  (sizeof(struct msgbuf)
                          -  sizeof(long));
    returnmsgctl (K, IPC_SET,  &bufp);
  >
  else
    return -1;
}
```

Jeżeli komunikat ma pustą treść, to kanał ma „nieskończoną" pojemność (w rzeczywistości jest ona ograniczona pewną stałą systemową). Jeżeli w takiej sytuacji chcemy ograniczyć pojemność kanału do ustalonej przez nas wartości, to musimy posługiwać się komunikatami o niepustej treści.

9.1.5 Ograniczenia

Kanały w Unixie przypominają nieco przestrzeń krotek w Lindzie. Jednak są one mechanizmem słabszym, ze względu na ograniczoną możliwość selektywnego wyboru, który sprowadza się tylko do pojedynczej liczby naturalnej (identyfikatora podkanahi). Czasami możnajednak obejść to ograniczenie, odwzorowując kombinacje wartości identyfikujących krotki w liczby naturalne (por. 9.3.6). Z drugiej strony, w operacji pobierania komunikatu można specyfikować zakres identyfikatorów podkanałów, co daje dodatkowe możliwości selektywnego wyboru, których nie ma w Lindzie.

9.2 Przykłady

9.2.1 Wzajemne wykluczanie

Implementacja semafora binarnego

Problem wzajemnego wykluczania można rozwiązać za pomocą komunikatów stosując je jako mechanizm synchronizacji. Kanał o pojemności jednego komunikatu będzie pełnił rolę

semafora binarnego, a liczba komunikatów w kanale będzie oznaczała wartość semafora. Treść komunikatu nie ma żadnego znaczenia (w tym rozwiązaniu jest to Uczba 0), znaczenie ma jedynie jego istnienie. Operację VB zaimplementowaUśmy tu jako funkcję logiczną, której wartość zależy od tego, czy udało się podnieść semafor.

```
 type komunikat  =  ...;
 var  K: channel;     {pojemność - jeden komunikat}

 procedure PB;
 var p:  integer;      {podkanal - nieistotny}
     k:  komunikat;    {komunikat - też nieistotny}
begin
   wez(K, 1, p, k)
end;   {PB}

function VB: boolean;
begin
   VB  := n_wloz(K,  1,  0)
end;   {VB}
```

W operacji weź jednoznacznie specyfikujemy podkanał, z którego ma być pobrany komunikat. Zatem wartością parametru p po powrocie z tej operacji będzie identyfikator właśnie tego podkanału. W takiej sytuacji parametr ten nie ma żadnego znaczenia.

Ponieważ początkowo kanał jest pusty, więc nie ma tu problemów z nieokreśloną wartością, tak jak w przypadku semaforów. Nie ma zatem znaczenia, kiedy wykona się po raz pierwszy procedura VB, umieszczająca komunikat w kanale.

Podajemy tu zapisany w języku C przykładowy kod trzech funkcji, które mogą być użyte do wzajemnego wykluczania. Funkcje deklaracja, PB oraz VB są zgrupowane w jednym pliku, który może być oddzielnie kompilowany i łączony z innymi plikami wynikowymi. Funkcje te działają na strukturach danych lokalnych dla tego pliku (klasa pamięci static).

Każdy proces, który chce korzystać z procedur PB i VB, musi najpierw wywołać procedurę deklaracja. W procedurze tej proces uzyskuje identyfikator kanału. Jeżeli kanał jeszcze nie istnieje, to jest tworzony. Ponadto jego pojemność jest zmniejszana do jednego komunikatu. Proces pełniący rolę inicjatora musi wywołać procedury deklaracja, a następnie VB, gdyż początkowo semafor musi mieć wartość 1.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
static int K;
static struct  {
   long podkanał;
   char c;
}  komunikat;

void  deklaracja  (void)
{
   struct msqid_ds bufp;
   K = msgget (1, IPC_CREAT  | 0666);
   komunikat.podkanał  =  1;
  msgctl (K, IPC_STAT,  &bufp);
  bufp.msg_qbytes =  1;
  msgctl (K, IPC_SET,  &bufp);
}

void PB (void)
```

```
{
  msgrcv (K, &komunikat,  1, 1, 0);
}

int  VB (void)
{
  return ( !msgsnd  (K,  &komunikat,  1,  IPC_NOWAIT));
}
```

Implementacja semafora dwustronnie ograniczonego

Implementacja semafora dwustronnie ograniczonego różni się od implementacji semafora binarnego przede wszystkim tym, że w kanale znajdować się może dowolna liczba komunikatów. Proces inicjujący musi wykonać tyle razy procedurę VD, ile ma wynosić wartość początkowa semafora. Poza tym operacja VD musi być blokująca. (Operacja PD jest taka sama, jak PB w implementacji semafora binarnego.)

```
procedure VD;
begin
  wloz(K, 1, 0)
end;   {VD}
```

9.2.2 Producenci i konsumenci

Rozwiązanie problemu producentów i konsumentów za pomocą mechanizmu komunikatów w Unixie jest bardzo proste, ponieważ mechanizm ten powstał specjalnie do tego typu problemów. Kanał w naturalny sposób spełnia tu rolę bufora. Należy jednak pamiętać o ograniczeniach związanych ze zwiększaniem pojemności kanału.

9.2.3 Czytelnicy i pisarze

Rozwiązanie z możliwością zagłodzenia pisarzy

Do synchronizacji procesów użyjemy jednego pustego komunikatu. Numer podkanału, w którym znajduje się ten komunikat, zmniejszony o 1 oznacza liczbę czytelników znajdujących się w czytelni. Zatem pisarz może wejść do czytelni tylko wtedy, gdy komunikat jest w podkanale numer 1. Aby zapewnić wzajemne wykluczanie, pisarz wchodzący do czytelni pobiera komunikat i zwraca go wychodząc z czytelni. Czytelnik natomiast podczas wchodzenia do czytelni przekłada komunikat z jakiegoś podkanału do podkanału o numerze o 1 większym, a przy wychodzeniu - do podkanału o
numerze o 1 mniejszym.

```
const C = ?;
      P = ?;
type komunikat =  integer;
var K:  channel;

process INICJATOR;
begin
  wloz(K, 1, 0)       {0  czytelników, czyli wstawienie do podkanału 1}
end;  {INICJATOR}

processCZYTELNIK(i:  l..C);
```

```
var c:  integer;       {podkanal =  liczba czytelników + 1}
    k:  komunikat;     {komunikat  - nieistotny}
begin
  while true do begin
    wlasne_sprawy;
    wez(K, 0, c, k);  {pobranie z dowolnego podkanału}
    wloz(K, c+l, 0);  {wstawienie do następnego podkanału}
    czytanie;
    wez(K, 0, c, k);  {pobranie z dowolnego podkanału}
    wloz(K, c-l, 0)   {wstawienie do poprzedniego}
  end
end;  {CZYTELNIK}

processPISARZ(i:  l..P);
var c:  integer;       {podkanal - nieistotny}
    k:  komunikat;     {komunikat - nieistotny}
begin
  while true do begin
    wlasne_sprawy;
    wez(K, 1, c, k);  {pobranie z podkanału 1}
    pisanie;
    wloz(K, 1, 0)     {wstawienie do podkanału 1}
  end
end;  {PISARZ}
```

Przedstawiamy również kod tego rozwiązania w języku C. Ponieważ treść komunikatu jest pusta, więc struktura do przekazywania komunikatów redukuje się do jednego pola, w którym przekazuje się identyfikator podkanału. Zamiast struktury można zatem użyć pojedynczej zmiennej. Wielkość komunikatu jest w takim rozwiązaniu równa zeru. Oto treść procesu:

```
INICJATOR:

   #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/msg.h>

   int K;
   long c;          /* zamiast struktury z komunikatem */

   void main  (void)
   {
     K = msgget  (1, IPC_CREAT  |  0666);
     c = 1;         /* 0 czytelników, czyli wstawienie do podkanału 1 */
     msgsnd (K,  &c, 0, 0);
   }

   Proces CZYTELNIK

   #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/msg.h>

   int K;
   long c;        /*podkanal = l.czytelników + 1 zamiast struktury, z komunikatem*/
   void main  (void)
   {
     K = msgget  (1, IPC_CREAT  |  0666);
     while (1) {
       wlasne_sprawy ();
```

```
      msgrcv(K, &c, 0, 0, 0);  /*wzięcie z dowol. podkanalu*/
      c++;                     /*zwiększenie nru podkanalu*/
      msgsnd(K, &c, 0, 0);     /*wstawienie do kanalu*/
      czytanie  ();
      msgrcv(K, &c, 0, 0, 0);  /*wzięcie z dowol. podkanalu*/
      c--;                     /*zmniejszenie nru podkanalu*/
      msgsnd(K, &c, 0, 0);     /*wstawienie do kanału*/
    }
  }

  Proces PISARZ

  #include <sys/tes.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>

  int K;
  long  c;                 /* zamiast struktury z komunikatem */
  void  main (void)
  {
    K = msgget (1, IPC_CREAT  |  0666);
    while  (1)  {
      wlasne_sprawy ();
      msgrcv(K, &c, 0, 1, 0);  /* pobranie z podkanału 1 */
      pisanie  ();
      msgsnd(K, &c, 0, 0);     /* wstawienie do podkanału 1 */
    }
  }
```

Rozwiązanie poprawne

W tym rozwiązaniu zmienne dzielone opisujące stan systemu będą umieszczone w jednym komunikacie, co zapewni wzajemne wykluczanie przy dostępie do nich. Komunikat ten będzie umieszczony w podkanale 1 przez proces INICJATOR.

Procesy czekają na możliwość pobrania komunikatu - sygnału umożliwiającego wejście do czytelni. Dla czytelników sygnałem jest komunikat w podkanale 2, a dla pisarzy - w podkanale 3. Komunikaty przesyłane tymi podkanałami pełnią rolę semaforów.

W celu zwiększenia czytelności kodxi zamiast identyfikatorów podkanałów będziemy używać stałych symbolicznych - stan dla określenia podkanału 1, start_czyt dla podkanału 2 i start_pis dla podkanału 3.

```
  const C  = ?;
        P  = ?;
        stan = 1;
        start_czyt = 2;
        start_pis   = 3;
type komunikat =
           record
             cc:  integer;    {1.  czekających czytelników}
             dc:  integer;    {1.  działających czytelników}
             cp:  integer;    {1.  czekających pisarzy}
             dp:  integer;    {1.  działających pisarzy}
           end;
    sygnał = integer;
var K:  channel;

process INICJATOR;
var k:  komunikat;
```

```
begin
  with k do begin
    cc := 0;
    dc := 0;
    cp := 0;
    dp := 0
  end
  wloz(K, stan,  k)
end;  {INICJATOR}

processCZYTELNIK(i:  l..C);
var p:  integer;                {podkanał -  nieistotny}
    s:  sygnał;                 {komunikat - też nieistotny}
    k:  komunikat;
begin
  while true do  begin
    wlasne_sprawy;
    wez(K, stan, p, k);
    if k.cp+k.dp >  0 then begin
       k.cc := k.cc + 1;        {są pisarze, trzeba czekać}
       wloz(K, stan,  k);
       wez(K, start_czyt, p,  s)
                               {czekanie}
    end else begin             {czytelnik wchodzi}
      k.dc := k.dc  + 1;
      wloz(K, stan, k)
    end;
    czytanie;
    wez(K, stan, p, k);
    k.dc := k.dc -  1;
    if (k.dc = 0) and (k.cp > 0)  then
      wloz(K, start_pis, s);   {wpuszczenie pisarza}
    wloz(K, stan, k)
  end
end;  {CZYTELNM}

processPISARZ(i: l..P);
var p: integer;                {podkanał -  nieistotny}
    s: sygnał;                 {komunikat - też nieistotny}
    k: komunikat;
    j: integer;
begin
  while true do begin
    wlasne_sprawy;
    wez(K, stan, p, k);
    if k.dc + k.dp > 0 then    {są działający czytelnicy]
    begin                      {lub pisarze}
      k.cp := k.cp + 1;         {trzeba czekać}
      wloz(K, stan, k);
      wez(K, start_pis,  p, s) {czekanie}
    end
    else begin                 {pisarz wchodzi}
      k.dp := 1;
      wloz(K, stan, k)
    end;
    pisanie;
    wez(K, stan, p, k);
    if k.cc > 0 then begin     {czekają czytelnicy}
      k.dp := 0;
      k.dc := k.cc;
      k.cc := 0;
```

```
      for j := 1 to k.cc do   {wpuszczenie wszystkich}
       wloz(K, start_czyt, s) {czytelników}
    else
    if k.cp > 0 then begin    {czekają pisarze}
      k. cp : = k. cp - 1;
      wloz(K, start_pis, s)   {wpuszczenie pisarza}
    end
    else k.dp  := 0;
    wloz(K, stan, k)
  end
end;  {PISARZ}
```

9.2.4 Implementacja monitora ograniczonego

Do synchronizacji procesów użyjemy komunikatu, pełniącego trzy funkcje. Po pierwsze, będzie to żeton uprawniający do korzystania z monitora, po drugie, będzie w nim przechowywana liczba procesów czekających na spełnienie warunkn (wstrzymanych po wait i czekających na signal), po trzecie będą w nim przechowywane wszystkie struktury danych monitora. Posiadaczem żetonu jest proces aktywny wykonujący procedury monitora. Jeżeli nie ma aktywnego procesu, to żeton znajduje się w podkanale 1, nazwanym tutaj wolny. Proces wchodzący do monitora pobiera żeton z tego podkanału i zwraca go, gdy wychodzi z monitora albo gdy zostaje wstrzymany w wyniku wykonania operacji wait. Proces wstrzymany oczekuje na żeton, który otrzyma z podkanału 2 (nazwanego tutaj budzenie) przez proces wykonujący operację signal.

```
const wolny = 1;
      budzenie  = 2;
type komunikat  = record
        w: integer;  {liczba procesów wstrzymanych po wait}
                      {struktury danych monitora}
     end;
var  K: channel;
    p: integer;       {podkanal - nieistotny}
    k: komunikat;

procedurę wejście;
begin
  wez(K,  wolny,  p,  k)
end;

procedure wait;
begin
  k.w :=  k.w+l;
  wloz(K,  wolny, k);
  wez(K,  budzenie,  p, k);
  k.w :=  k.w-1
end;

procedurę signal;
begin
  if k.w  > 0  then wloz(K, budzenie,  k)
              else wloz(K, wolny,  k)
end;
function empty: boolean;
begin
  empty  := k.w = 0
end;
```

```
procedure wyjście;
begin
  wloz(K, wolny, k)
end;

process  INICJATOR;
var k: komunikat;
begin
  k.w  := 0;
  wloz(K, wolny, k)
end;   {INICJATOR}
```

Kolejność wznawiania procesów wstrzymanych po operacji wait jest w tym rozwiązaniu nieokreślona.

9.3 Zadania

9.3.1 Implementacja semafora typu OR

Zaimplementuj semafor typu OR za pomocą operacji na kanałach w systemie Unix.

9.3.2 Implementacja monitora ogólnego

Zaimplementuj monitor ogólny za pomocą operacji na kanałach w systemie Unix.

9.3.3 Problem podziału

Rozwiąż zadanie 5.3.3 za pomocą operacji na kanałach w systemie Unix.

9.3.4 Zasoby dwóch typów

Rozwiąż zadanie 4.3.10 za pomocą operacji na kanałach w systemie Unix.

9.3.5 Lotniskowiec

Rozwiąż zadanie 3.3.11 za pomocą operacji na kanałach w systemie Unix.

9.3.6 Głosowanie

Rozwiąż zadanie 5.3.6 za pomocą operacji na kanałach w systemie Unix.

9.3.7 Komunikacja przez pośrednika

Rozwiąż zadanie 5.3.7 za pomocą operacji na kanałach w systemie Unix.

9.4 Rozwiązania

Rozwiązania zadań podajemy używając, abstrakcyjnej notacji. Czytelnika zachęcamy do zapisania ich również w języku C, z wykorzystaniem funkcji systemu Unix.

9.4.1 Implementacja semafora typu OR

Dzięki możliwości wyspecyfikowania podkanału, z którego ma być pobrany komunikat, rozwiązanie tego zadania za pomocą mechanizmu komunikatów w Unixie jest bardzo proste. Wystarczy dla każdego semafora zarezerwować jeden podkanał w tym samym kanale i podczas wykonywania operacji POR pobierać komunikat z dowolnego z tych podkanałów. Operacje P i V na pojedynczych semaforach są zaimplementowane podobnie, jak w p. 8.2.1. Parametrem operacji jest tu identyfikator podkanału odpowiadającego semaforowi, na którym ta operacja ma być wykonana.

```
type komunikat = integer;
var K:  channel;
procedure POR(var  i: integer);
var k:  komunikat;                {komunikat - nieistotny>
begin
  wez(K,  0,  i, k)
end;

procedure P(s: integer);
var k:  komunikat;                {komunikat - nieistotny}
    i:  integer;                  {podkanal - nieistotny}
begin
  wez(K,  s,  i, k)
end;

procedure V(s: integer);
begin
  wloz(K,  s,  0)
end;
```

W operacji weź w procedurze POR jako identyfikatora podkanału użyliśmy liczby 0. Dzięki temu oba semafory są wykorzystywane równomiernie. Osiągnięcie tego efektu w rozwiązaniu zadania 3.3.5 było dosyć trudne. Użycie liczby -2 powodowałoby wykorzystywanie głównie semafora reprezentowanego przez pierwszy podkanał.

Implementacja semaforów typu AND za pomocą komunikatów jest o wiele bardziej skomplikowana. Mechanizm komunikatów nie dostarcza żadnych ułatwień do synchronizacji typu AND i dlatego musi być ona oparta na rozwiązaniu zadania 3.3.4. W razie konieczności stosowania synchronizacji typu AND lepiej jest zatem używać semaforów unixowych.

9.4.2 Implementacja monitora ogólnego

Prezentowane rozwiązanie stanowi pełną implementację monitora. Jest możliwe wykonywanie operacji signal w dowolnym miejscu procedury monitora, a ponadto procesy wstrzymane po wykonaniu operacji signal są wznawiane w kolejności odwrotnej niż były wstrzymywane. Dodatkowo zaimplementowaliśmy C zmiennych typu condition. Są one identyfikowane kolejnymi liczbami naturalnymi.

Stan oraz struktury danych monitora są przechowywane w jednym komunikacie pełniącym rolę żetonu. Posiadacz komunikatu jest procesem aktywnym w monitorze. Jeżeli nie ma procesu aktywnego w monitorze, to komunikat znajduje się w podkanale wolny kanału K1.

Każdy proces wstrzymany po wykonaniu operacji signal czeka na komunikat w osobnym podkanale o identyfikatorze po_signal+k.s, przy czym k.s jest liczbą procesów wstrzymanych w danej chwili, czyli numerem kolejnego wstrzymanego procesu. Operacja wznawiania takich procesów dotyczy procesu o największym w danej chwili numerze. Wznawianie odbywa się zatem w kolejności odwrotnej niż wstrzymywanie, czyli zgodnie z definicją monitora.

Osobny kanał K2 rezerwujemy do wstrzymywania procesów wykonujących operację wait. Do pamiętania liczby procesów wstrzymanych po operacji wait w kolejce odpowiadającej zmiennej typu condition o numerze i służy element i tablicy w. Procesy są wstrzymywane w oczekiwaniu na komunikat w podkanałach o identyfikatorach od N*(i-l)+l do N*i (zakładamy, że będzie co najwyżej N takich procesów i pomijamy kontrolę). Pierwszy podkanał, przy którym oczekuje proces, ma identyfikator k.p[i], a ostatni k.k[i]. Identyfikatory podkanałów są zwiększane modulo N, czekające procesy tworzą więc kolejkę cykliczną.

```
   const  N = ?;          {liczba zmiennych condition}
          C = ?;          {maksymalna liczba procesów}
          wolny =  1;
      po_signal = 1;
type
komunikat = record
     s: integer;    {1.  procesów wstrzymanych po signal}
     w: array[l..C]  of integer;
                    {1.  procesów wstrzymanych po wait}
     p, k: array[l..C] of integer;
                    {początki i końce kolejek procesów czekających na signal}
                    {struktury danych monitora}
end;
var K1: channel;
    K2: channel;
    p: integer;     {podkanał -  nieistotny}
    k: komunikat;

procedurę wejście;
begin
  wez(Kl, wolny, p,  k)
end;

procedure wait(i: l..C);
begin
  k.w[i]  := k.w[i] + l;
  k.p[i]  := k.p[i] mod N + 1 + N *  (i-l);
  if k.s > 0 then wloz(Kl, po_signal+k.s,  k)
             else wloz(Kl, wolny, k);
  wez(K2, k.p[i], p,  k);
  k.w[i]  := k.w[i]-l
end;
```

```
procedure signal(i:  l..C);
begin
  if k.w[i] > 0 then  begin
    k.s := k.s+l;
    k.k[i] := k.k[i]  mod N + 1 + N  * (i-l);
    wloz(K2, k.k[i],  k);
    wez(Kl, po_signal+k.s, p, k);
    k.s := k.s-1
  end
end;
   function empty(i:  l..C):  boolean;
   begin
     empty := k.w[i]  =  0
   end;

   procedure signal_i_wyjscie;
   begin
     if k.w[i]  > 0 then begin
       k.k[i]  := k.k[i] mod  N + 1 + N * (i-l);
       wloz(K2, k.k[i], k)
     else
     if k.s > 0 then  wloz(Kl, po_signal+k.s, k)
                else  wloz(Kl, wolny,  k)
   end;

   procedure wyjście;   {gdy na końcu nie ma signal}
   begin
     if k.s > 0 then  wloz(Kl, po_signal+k.s, k)
                else  wloz(Kl, wolny,  k)
   end;

   process INICJATOR;
   var k: komunikat;
       i: integer;
   begin
     k.s  := 0;
     for  i := 1 to C  do begin
        k.w[iJ  := 0;
        k.p[i]  := N*i;
        k.k[i]  := N*i
     end
     wloz(Kl, wolny,  k)
   end;   {INICJATOR}
```

9.4.3 Problem podziału

Rozwiązanie tego zadania za pomocą komunikatów jest podobne do rozwiązania w p. 5.4.3. Różnice polegają na jawnym kończeniu procesu TPROC oraz na przekazywaniu liczb przez kanał. Do przekazywania liczb od procesu SPROC do TPROC użyjemy podkanału 1, a w kierunku odwrotnym podkanału 2.

Procesy wykonują na przemian operacje weź i włóż, więc w kanale K może być zawsze co najwyżej jedna liczba. Jednak musimy użyć dwóch podkanałów, aby proces, który właśnie umieścił liczbę w kanale, nie pobrał jej natychmiast sam.

```
   const S =  ?;
         T =  ?;
   type komunikat =  integer;
   var K: channel;
```

```
process SPROC;
var zbiórS:  array[l..S] of  integer;
    m, n:  komunikat;
    dalej : boolean;
    p: integer;
begin
  dalej := true;
  while dalej do  begin
    m := MaxElem(zbiorS);
    wloz(K,  1, m);
    wez(K, 2, p,  n);
    if m <>  n then
      zbiórS := zbiórS - {m}  +  {n}
    else dalej  := false
  end
end;  {SPROC}

process TPROC;
var zbiórT:  array[l..T] of  integer;
    m, n:  komunikat;
    dalej : boolean;
    p: integer;
begin
  dalej := true;
  while dalej do  begin
    n := MinElem(zbiórT);
    wez(K, 1, p,  m);
    wloz(K,  2, n);
    if m <>  n then
      zbiórT := zbiórT + {m}  -  {n>
    else dalej  := false
  end
end;  {TPROC}
```

9.4.4 Zasoby dwóch typów

Każdemu typowi zasobu przyporządkujemy odrębny podkanał. Każdy zasób będzie reprezentowany przez komunikat z numerem zasobu, umieszczony w odpowiednim podkanale. Pobranie komunikatu będzie oznaczać uzyskanie prawa dostępu do zasobu. Po zakończeniu korzystania z zasobu proces będzie zwracał komunikat do podkanału. Proces INICJATOR umieszcza na początku odpowiednie komunikaty w podkanałach.

```
const A =  1;
      B = 2;
      M = ?;
      N = ?;
type komunikat = integer;
var K: channel;

process INICJATOR;
var i: integer;
begin
  for i := 1 to M do  wloz(K,  A,  i);
  for i := 1 to N do  wloz(K,  B,  i)
end;  {INICJATOR}

process GRUPA1;
```

```
var p: integer;   {podkanal  - nieistotny}
    k: komunikat;
begin
  while true do begin
    wlasne_sprawy;
    wez(K, A, p, k);
    korzystaj(A, k);
    wloz(K, A, k)
  end
end;  {GRUPA1}

process GRUPA2;
var jaki: A..B;
    k: komunikat;
begin
  while true do begin
    wlasne_sprawy;
    wez(K, -2, jaki,  k);
    korzystaj(jaki, k);
     wloz(K, jaki,  k)
   end
 end;   {GRUPA2}

 process GRUPA3;
 var jaki :  A..B;
     k:  komunikat;
 begin
   while true do begin
     wlasne_sprawy;
     if  n_wez(K, 0,  jaki, k) then  begin
       korzystaj(jaki, k);
       wloz(K, jaki,  k)
     end
   end
 end;   {GRUPA3}
```

Proces grupy drugiej podczas pobierania komunikatii podaje liczbę ujemną jako identyfikator podkanału. Dzięki temu, jeżeli tylko w podkanale numer 1 reprezentującym wygodniejszy zasób będzie komunikat, to zostanie on pobrany w pierwszej kolejności. Natomiast jeśli proces grupy trzeciej chce pobrać komunikat, podaje liczbę 0 jako identyfikator podkanału. To powoduje pobieranie komunikatu najdawniej umieszczonego w kanale i w rezultacie równomierne wykorzystanie obu typów zasobów.

9.4.5 Lotniskowiec

Do opisywania stanu systemu użyjemy jednego komunikatu. Będzie on znajdował się w podkanale rejestr. Musi go tam umieścić proces INICJATOR. Samolot, który chce startować lub lądować, wykonuje procedurę chcę. Procedura ta pobiera konumikat, zmienia odpowiednio informację w nim zawartą i zwraca go do podkanału. Następnie, jeżeli nie może natychmiast skorzystać z pasa startowego, czeka na sygnał zezwalający na start lub lądowanie. Sygnał zezwalający na start przesyła się podkanałem startuj, a na lądowanie - podkanałem ląduj. Sygnały te są wysyłane przez samolot, który przestaje korzystać z pasa startowego po wykonaniu procedury koniec.

```
const N =  ?;
      K =  ?;
```

```
         rejestr = 1;
         startuj  = 2;
         ląduj = 3;
type komunikat = record
        jest:  integer;  {1.  samolotów na lotniskowcu}
        start: integer;  {1.  chcących startować>
        lądow: integer;  {1.  chcących lądować}
        wolny: boolean;  {czy wolny pas}
     end;
     operacja = (start, lądow);
     sygnał = integer;

var C:  channel;

procedure chce(op: operacja);
var p:  integer;          {podkanal - nieistotny}
    k:  komunikat;
    w:  boolean;
    s:  sygnał;
begin
  wez(C, rejestr,  p, k);
  if op = start then k.start  :=  k.start+1
                else k.lądow  :=  k.lądow+1;
  w := k.wolny;
  if w and ((op =  start) or (k.jest < N))
  then k.wolny :=  false;
  wloz(C, rejestr, k);
  if not w or ((op = lądow) and  (k.jest = N))
  then
  if op = start then wez(C, startuj, p, s)
                else wez(C, ląduj, p, s)
end;  {chce}

procedure koniec(op: operacja);
var p:  integer;          {podkanal - nieistotny}
    k:  komunikat;
    s:  sygnał;
begin
  wez(C, rejestr,  p, k);
  if op = start then begin
    k.start := k.start-1;
    k.jest  := k.jest -1
  end
  else begin
    k.lądow := k.lądow-1;
       k.jest   := k.jest +1
     end;
     if k.jest < K then begin
       if  k.lądow > 0 then wloz(C, ląduj,  s)
       else
       if  k.start > 0 then wloz(C, startuj,  s)
       else  k.wolny := true
     end
     else  begin
       if  k.start > 0 then wloz(C, startuj,  s)
       else
       if  (k.lądow > 0) and (k.jest < N) then
         wloz(C, ląduj, s)
       else  k.wolny := true
     end;
     wloz(C, rejestr, k)
```

```
end;  {koniec}

process INICJATOR;
var k:  komunikat;
begin
  k.jest  := ?;
  k.start := 0;
  k.lądow := 0;
  k.wolny := true
end;  {INICJATOR}
```

9.4.6 Głosowanie

Rozwiązanie to jest wzorowane na rozwiązaniu z p. 7.4.3. Nie mamy jednak operacji odczytywania komunikatu bez pobierania go z kanału, podobnej do operacji READ w Lindzie, dlatego każdy proces musi umieścić w kanale swój głos N razy. Poza tym w rozwiązaniu w Lindzie krotka z głosami była identyfikowana dwiema liczbami: numerem tury i numerem procesu. Tutaj każdy komunikat będzie przesyłany odrębnym podkanałem, identyfikowanym kombinacją numeru tury i numeru procesu,

```
const N = ?;
var K:  channel;

process P(i:  1..N);
var t:  array[l..N]  of integer;
    tura, min,  j, k,  pk, g: integer;
begin
  for  j  :=  1  to N do t[j] := 1;
  tura := 0;
  repeat
    tura  := tura+l;
    g  :=  glosuj(t);
    for j  :=  1  to N  do
       wloz(K, tura*N+i, g);
    for j  :=  1  to N  do t[j]  := 0;
    for j  :=  1  to N  do begin
       wez(K,  tura*N+j, pk, k);
       t[k]  := t[k]+l
    end;
    min  : =  1;
    while t[min]  = 0 do   {szukanie niezerowego minimum}
       min := min + 1;
    for j  := min+1 to N do
       if  (O<t[j]) and (t[j]<t[min]) then min  := j;
    koniec  := t[min] = N;
    t[min]  := 0
  until koniec
end;   {P}
```

9.4.7 Komunikacja przez pośrednika

Rozwiązanie tego problemu za pomocą mechanizmu komunikatów w Unixie jest bardzo proste. Rolę pośrednika może spełniać kanal, w którym dla każdego procesu będzie istniał odrębny podkanał. W kanale trzeba umieścić początkowo M komunikatów. Zrobi to proces INICJATOR.

```
   const  N = ?;
   var K: channel;

   process INICJATOR;
   var i, j : integer;
       k: komunikat;
   begin
     for  i  :=  1 to M do begin
       losuj(j) ;
       k  :=  nowy_komunikat;
       wloz(K, j,  k)
     end
   end;   {INICJATOR>
process P(i:  l..N);
var j:  integer;
    k:  komunikat;
begin
  while true  do  begin
    wez(K, i, j, k);
    przetwórz(k);
    losuj(j);
    wloz(K, j, k);
    wlasne_sprawy
  end
end;  {P}
```

10 Zdalne wywołanie procedur w systemie Unix

10.1 Wprowadzenie

10.1.1 Idea zdalnego wywołania procedury

Zdalne wywołanie procedury w Unixie [Berk86, LFJL86, SimM88, AT&T90a] będziemy dalej przedstawiać korzystając z abstrakcyjnej, wysokopoziomowej notacji. Podamy jednak również funkcje systemowe, realizujące ten mechanizm. Ponadto niektóre z przykładów zapiszemy w języku C, korzystając z tych funkcji.

Zdalne wywołanie procedury (RPC - Remote Procedure Call) przypomina wywołanie procedur monitora. Monitorowi w tym przypadku odpowiada proces, nazywany procesem obslugującym (server), udostępniający procedury zdalne, które mogą być wywołane przez inne procesy, nazywane procesami wolającymi (clients). Podobnie, jak w monitorze, jest zapewnione wzajemne wykluczanie procesów podczas wykonywania procedur zdalnych. Jednak w przeciwieństwie do procedur monitora lub zwykłych procedur, procedury zdalne są wykonywane w przestrzeni adresowej procesu obsługującego, a nie procesu wołającego - stąd bierze się nazwa tego mechanizmu. Poza tym w procesie obsługującym nie ma odpowiednika typu kolejkowego monitora oraz związanych z nim operacji.

Jedyne co wyróżnia proces obsługujący, to fakt udostępniania swoich procedur. Zatem każdy proces, który ma do zaoferowania innym jakieś usługi, może być procesem obsługującym. Co więcej, ten sam proces może być zarówno procesem obsługującym, jak i wołającym.

Funkcjonalnie mechanizm RPC jest na tyle podobny do monitora, że dalej będziemy stosować abstrakcyjną notację podobną do notacji z rozdz. 4.
Jedynie słowo kluczowe monitor zastąpimy słowem process i wprowadzimy nową notację dla specjalnego trybu zdalnego wywołania procedur nazywanego rozgłaszaniem. Zapis mechanizmu RPC w języku C z użyciem funkcji systemu Unix jest całkiem inny. Sposób przekształcenia notacji abstrakcyjnej w kod w języku C podajemy w dalszej części tego rozdziału, w której przedstawiamy oryginalne funkcje realizujące zdalne wywołanie procedur, wprowadzone po raz pierwszy w systemie Unix BSD. Później zdalne wywołanie procedur zostało zaimplementowane również w innych wersjach systemu Unix, częstojednak funkcje realizujące ten mechanizm różnią się nieco od oryginalnych. Opisujemy tu tylko kilka funkcji tworzących najwyższą warstwę zdalnego wywołania procedur. Ich znajomość całkowicie wystarczy do pisania programów rozproszonych. Oprócz nich istnieje jeszcze kilkadziesiąt innych funkcji, mają jednak one znaczenie pomocnicze (na przykład pozwalają zwiększyć efektywność programu rozproszonego oraz umożliwiają szyfrowanie przekazywanych danych).

10.1.2 Reprezentacja danych

Zazwyczaj procesy wołające i proces obsługujący są wykonywane przez różne komputery połączone siecią komunikacyjną, ale nic nie stoi na przeszkodzie, aby był to jeden komputer. Korzystanie z mechanizmu RPC w obu przypadkach wygląda tak samo. Jednak

jeżeli proces obsłujący i procesy wołające są realizowane przez komputery, w których dane są reprezentowane w różny sposób, to zachodzi konieczność konwersji parametrów wejściowych i wynikowych procedur zdalnych do pewnej jednolitej postaci. Standardem stosowanym w systemie Unix jest XDR -eXtended Data Representation [SunM87]. Podczas zdalnego wywołania procedury parametry wejściowe są przekształcane z reprezentacji komputera wykonującego proces wołający do postaci XDR. W tej postaci są one przekazywane przez sieć do komputera wykonującego proces obsługujący, a tam są przekształcane do reprezentacji obowiązującej dla tego komputera. Wyniki podlegają przekształceniu odwrotnemu.

Konwersja odbywa się przez specjalne funkcje, nazywane filtrami. Zarówno przekształcenie do postaci XDR, jak i przekształcenie odwrotne dokonuje się przez ten sam filtr, który rozpoznaje kierunek przekształcenia. Niezależnie od tego, czy konwersja jest potrzebna, czy też nie (na przykład podczas realizacji procesu obsługującego i procesu wołającego przez ten sam komputer), kod programu w języku C ze zdalnym wywołaniem procediir zawsze zawiera wywołanie filtrów. Dzięki temu osiągniętojednolity, przenośny mechanizm.

Wartością filtrów jest 0, gdy proces konwersji się nie powiódł, albo liczba różna od zera w przeciwnym razie. Dla podstawowych typów języka C istnieją predefiniowane filtry XDR. Oto niektóre z nich:

```
xdr_int (dla typu int),
xdrjfloat (dla typu float),
xdr_char (dla typu char),
xdr_wrapstring (dla typu char *).
```

Ponadto zdefiniowano typ logiczny bool_t i filtr xdr_bool. W przypadku struktur filtry konstruuje programista. Na przykład dla struktury:

```
struct s {
  int  i;
  float x;
};
```

można zdefiniować następujący filtr:

```
int xdr_s (XDR *xdrsp, struct  s  *sp)
{
  return  (xdr_int (xdrsp, &sp->i)  &&
           xdr_float (xdrsp, &sp->x));
}
```

Znaczenie pierwszego parametru możemy tu pominąć - wystarczy jedynie wiedzieć, że w każdym filtrze musi on wystąpić i być użyty w taki sposób, jak w przedstawionym przykładzie. Drugim parametrem musi być zawsze wskaźnik.

Jeżeli procedura zdalna nie ma parametrów wejściowych lub wynikowych, to jako odpowiedniego filtru używa się xdr_void.

10.1.3 Proces obsługujący

Aby proces obsługujący mógł udostępniać procedury, do jego programu źródłowego trzeba dyrektywą #include włączyć plik rpc/rpc.h. Każda procedura zdalna procesu obsługującego jest identyfikowana zewnętrznie trzema liczbami naturalnymi: numerem

programu, numerem wersji, numerem procedury. Numer programu jest identyfikatorem procesu obsługującego. Musi on być unikalny w zbiorze numerów dla danego komputera i musi być liczbą szesnastkową z zakresu [0x20000000, Ox3FFFFFFF]. Jeden program może mieć kilka wersji.

Aby procedura procesu obsługującego mogła być wywołana, musi być najpierw zarejestrowana, w wyniku wywołania funkcji

```
int  registerrpc
    (unsigned long PROG, unsigned long VERS,
     unsigned long PROC,  char  *  (* proc)  (),
     int (* xdr_arg)  (), int  (* xdr_res)  ());
```

Parametry PROG, VERS i PROC oznaczają odpowiednio numer programu, wersji i rejestrowanej procedury. Parametr proc jest wskaźnikiem do funkcji, zdefiniowanej w procesie obsługującym, odpowiadającej rejestrowanej procedurze. Parametry xdr_arg i xdr_res są wskaźnikami do filtrów odpowiednio dla parametrów wejściowych i wynikowych rejestrowanej procedury. Jeżeli procedura została zarejestrowana, to wartością funkcji jest 0. Inna wartość oznacza, że procedura nie została zarejestrowana.

Dla kadej rejestrowanej procedury w procesie obsługiijącyni trzeba zdefiniować funkcję typu char *. Funkcja ta musi być albo bezparanietrowa, albo miećjeden parametr, który musi być wskaźnikiem. Zatemjeżeli chcemy przekazywać do niej więcej niż jeden parametr, to wszystkie parametry trzeba umieścić w jednej strukturze i do definiowanej funkcji przekazywać wskaźnik do tej struktury. Podobnie postępujemy z wynikami. Jeżeli funkcja nie ma wyniku, to jej wartością powinien być wskaźnik pusty rzutowany na wskaźnik znakowy

```
return (char *)NULL;
```

Po zarejestrowaniu wszystkich udostępnianych procedur i wykonaniu instrukcji inicjujących, w procesie obsługującym musi być wywołana funkcja bezparametrowa

```
void svc_run (void);
```

W funkcji tej proces obsługujący czeka nieaktywnie na wywołanie udostępnianych przez siebie procedur. Sterowanie nigdy z niej nie wraca. Zatem proces obsługujący jest procesem nieskończonym.

Następujący przykład wyjaśnia sposób rejestracji procedur zdalnych. Zakładamy, że proces obsługujący udostępnia procedurę CZAS. Oto zapis w notacji abstrakcyjnej:

```
processPRZYKLAD(i:  l..N);
type tt = record
                      {struktura do przechowywania czasu}
          end;

export procedure  CZAS(var t: tt);
begin
  t  := ...           {odczytanie bieżącego czasu}
end;   {CZAS>
begin
end;  {PRZYKLAD}
```

A oto kod programu w języku C:

```
#include <rpc/rpc.h>

#define  PRZYKŁAD 0x20000000
#define  WERSJA      1
#define  CZAS        1

typedef  struct {
```

```
                    /* struktura do przechowywania czasu */
} tt;

extern void podaj_czas (tt *);

int xdr_t (XDR *xdrsp, tt *tp)
{
                    /* filtr dla struktury t */
}

char  *czas  (void)
{
  static tt t;

  podaj_czas (&t);  /* odczytanie bieżącego czasu */
  return (char *)&t;
}

void  main (void)
{
  registerrpc (PRZYKŁAD, WERSJA, CZAS,
               czas,  xdr_void, xdr_t);
  svc_run ();
}
```

W funkcji czas zmienna t musi być zdefiniowana jako zmienna statyczna (klasa pamięci static). Jest to konieczne, gdyż filtr dla wyniku funkcji dziala dopiero po zakończeniu jej wykonywania i usunięciu jej zrębu ze stosu. Jeżeli nie byłaby to zmienna statyczna, to w obszarze pamięci zajmowanym uprzednio przez tę zmienną mogłoby znajdować się już coś całkiem innego. Wartością funkcji musi być adres zmiennej t, rzutowany na wskaźnik do zmiennej znakowej.

10.1.4 Proces wołający

Aby proces wołający mógł wywołać procedury zdalne, do jego programu źródłowego trzeba dyrektywą #include włączyć plik rpc/rpc.h. Proces wołający żąda wykonania procedury zdalnej wołając funkcję:

```
int callrpc (char *server, unsigned long PROG, unsigned long VERS, unsigned
         long  PROC),
             int  (*  xdr_arg)O,  char *arg,
             int  (*  xdr_res)O,  char *res);
```

Parametr server jest nazwą komputera, na którym działa proces obsługujący. Parametry PROG, VERS oraz PROC są odpowiednio numerami programu, wersji i wywołanej procedury. Parametry xdr_arg i xdrjres są filtrami odpowiednio dla parametru wejściowego i wynikowego wołanej funkcji, natomiast arg i res są adresami parametru wejściowego i wynikowego. Jeżeli wywołanie procedury powiodło się, to wartością funkcji jest 0. Inna wartość oznacza, że wystąpił jakiś błąd, na przykład procedura o podanych numerach nie była zarejestrowana.

Warto podkreślić, że w wywołaniu funkcji callrpc trzeba podać nazwę komputera wykonującego proces obsługujący. W dalszych rozwiązaniach podawanych w formie abstrakcyjnej będziemy ten aspekt pomijać, gdyż procesy będą identyfikowane jedynie swoją

nazwą i ewentualnie indeksem. Natomiast przedstawiając rozwiązania w języku C wskażemy sposób podawania nazwy komputera.

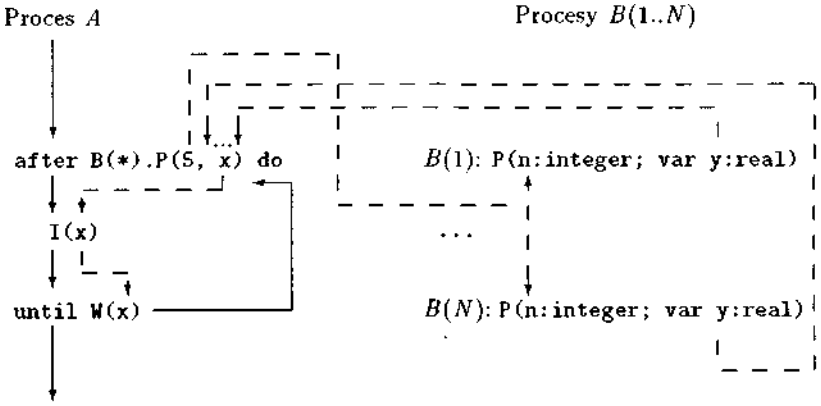
Jeżeli omówiony wcześniej proces obsługujący PRZYKŁAD będzie wykonywany up. przez komputer o nazwie nwsl850, to procedura CZAS może być wywołana w następujący sposób:

```
callrpc ("nwsl850", PRZYKŁAD, WERSJA, CZAS,
         xdr_void, NULL, xdr_t, &t);
```

przy czym PRZYKŁAD, WERSJA, CZAS są stałymi, które są równe odpowiednio 0x20000000, 1, 1, a t jest zmienną typu tt.

10.1.5 Rozgłaszanie

W systemie Unix można jednocześnie wywołać zdalne procedury, wykonywane przez różne procesy obsługujące. Metoda ta nazywa się rozylaszaniem (broadcasting) od sposobu jej realizacji. Polega ona na wysłaniu jednego komunikatu do wszystkich procesów obsługujących, powodującego wykonanie procedur zdalnych (rys. 10.1). Wykonanie może odbywać się równocześnie. Po rozgłoszeniu proces rozgłaszający czeka na zakończenie wykonania zdalnych procedur. Gdy któraś z nich zakończy się, w procesie wołającym jest wykonywany ciąg instrukcji (być może pusty) podany przez użytkownika. W instrukcjach tych można korzystać z wyników otrzymanych z właśnie zakończonej procedury. Następnie jest sprawdzany warunek logiczny również podany przez użytkownika. Jeśli ten warunek jest spełniony, to proces rozgłaszający przechodzi do wykonywania następnej instrukcji po instrukcji rozgłaszania. Wyniki wykonania procedur, które zakończą się później, będą wówczas zignorowane. W przeciwnym razie proces czeka na zakończenie się kolejnej procedury.



Rys. 10.1. Rozgłaszanie

Zaletą rozgłaszania, w porównaniu z sekwencyjnym zdalnym wywołaniem tej samej procedury w wielu procesach obsługujących, jest równoczesne wykonywanie procedur. Przy dużej liczbie procesów obsługujących może to znacznie zmniejszyć czas wykonania.

W abstrakcyjnej notacji rozgłaszanie będziemy zapisywać w postaci specjalnej instrukcji (podobnej do instrukcji pętli):

```
after S(*).P(a) do
   I
until W
```

w której S jest nazwą tablicy procesów obsługujących, * oznacza, że wywołanie dotyczy wszystkich procesów w tablicy, P jest nazwą procedyry zdalnej, a - listą parametrów aktualnych, I - ciągiem instrukcji wykonywanych po każdym zakończeniu procedury zdalnej, a

W - wyrażeniem logicznym obliczanym po każdym wykonaniu instrukcji I. Zarówno w instrukcjach,jak i w wyrażeniu można korzystać z wyników wykonań procedury P.

Wszystkie procedury zdalne, które mają być wywołane za pomocą rozgłaszania, muszą być zarejestrowane pod tym samym numerem programu, wersji i procedury. Wywołanie za pomocą rozgłaszania realizuje w systemie Unix funkcja:

```
int  clnt_broadcast
    (unsigned  long PROG, unsigned long VERS,
     unsigned  long PROC, int  (* xdr_arg)Q,  char *arg,
     int (* xdr_res)O, char  *res, int (*  res_proc)O);
```

Znaczenie wszystkich parametrów, z wyjątkiem ostatniego, jest takie, jak w funkcji callrpc. Ostatnim parametrem jest wskaźnik do funkcji definiowanej przez użytkownika, zawierającej instrukcje wykonywane po zakończeniu kolejnej procedury zdalnej

```
int  res_proc (char  *res, struct sockaddr_in *addr);
```

Pierwszym parametrem funkcji res_proc jest wskaźnik do wyników wykonaniaprocedury zdalnej, ten sam, który był parametrem funkcji rozgłaszającej clnt_broadcast. Konwersję wyników realizuje filtr, również bądący parametrem tej funkcji. Drugi parametr jest dla nas nieistotny. Jeżeli wartością funkcji res_proc jest FALSE (liczba 0), to proces wykonujący funkcję clnt_broadcast czeka na zakończenie kolejnej procedury zdalnej. Jeżeli wartością jest TRUE, to funkcja clnt_broadcast kończy się.

Wartością funkcji clnt_broadcast jest 0, jeżeli rozgłaszanie powiodło się, to znaczy przynajmniej jedna wywołana procedura zdalna się wykonała. Inna wartość oznacza, że wystąpił jakiś błąd, na przykład procedura o podanych numerach nie była nigdzie zarejestrowana. (W rzeczywistości semantyka funkcji clnt_broadcast jest nieco bardziej skomplikowana niż tu opisano.)

Podamy przykład procesu, w którym procedura CZAS jest wywołana za pomocą rozgłaszania. Natychmiast po otrzymaniu wyniku tej procedury CZAS, której wykonanie zakończyło się pierwsze, proces rozgłaszający przejdzie do realizacji procedury korzystaj (zakładamy t\i, że działa przynajmniej jeden proces obsługujący, udostępniający procedurę CZAS). Oto zapis w notacji abstrakcyjnej:

```
process P;
begin
  afterPRZYKLAD(*).CZAS(t) do   {nic}
  until true;       {czekamy na pierwszą odpowiedź}
  korzystaj(t)
end;  {P}
```

A oto zapis w języku C:

```
#include <rpc/rpc.h>
#define PRZYKŁAD  0x20000000
#define WERSJA       1
#define CZAS         1
typedef struct {
   /*  struktura do przechowywania czasu  */
}  tt;

int xdr_t (XDR *xdrsp,  tt *tp)
{
   /*  filtr dla struktury t */
}
```

```
extern void korzystaj  (tt *);

int po_rozgloszeniu  (char *t, struct sockaddr_in *addr)
{
   return TRUE;        /* powrót po pierwszej  odpowiedzi */
}

void  main (void)
{
   clnt_broadcast (PRZYKŁAD, WERSJA, CZAS,
               xdr_void, NULL, xdr_t, &t, po_rozgloszeniu);
   korzystaj (&t);
}
```

10.1.6 Ograniczenia

Istotnym ograniczeniem zdalnego wywołania procedur jest brak wbudowanego mechanizmu wstrzymywania procesów, podobnego do typu kolejkowego monitora. Ograniczenie to można obejść, używając omówionych dotychczas wysokopoziomowych mechanizmów synchronizacji (por. przykład 10.2.1 „Implementacja zdalnego semafora"). Alternatywnym rozwiązaniem jest modyfikacja mechanizmu zdalnego wywołania procedur. Wymaga to jednak wykorzystywania niskopoziomowych funkcji systemowych Unixa.

Wadą zdalnego wywołania procedur w wersji Sun RPC jest blokowanie procesu obsługującego podczas wykonywania procedury. Alternatywne rozwiązanie (zastosowane w np. w DCE RPC) dopuszcza równoczesne wywoływanie procedur w procesie obsługującym przez wiele procesów wołających. Jest wtedy konieczna synchronizacja dostępu do zmiennych dzielonych procesu obsługującego. (Zauważmy, że to podejście umożliwia łatwe ominięcie pierwszego ograniczenia.)

Innym ograniczeniem jest konieczność jawnego podawania nazwy komputera wykonującego proces obsługujący. Nie znaczy to co prawda, że użytkownik musi z góry wiedzieć, gdzie znajduje się proces obsługujący. Nazwę komputera wykonującego proces obsługujący o danym numerze programu i numerze wersji można otrzymać, korzystając z programu obsługującego rpc_bind. To ograniczenie jest uznawane za dużą wadę zdalnego wywołania procedur w systemie Unix [Sant91]. W różnych implementacjach tego mechanizmu wiele uwagi poświęca się problemowi automatycznej identyfikacji komputera wykonującego proces obsługujący. Przykładem może być projekt standardu zdalnego wywołania procedur - Distributed Computing Environment [RoKF92, Shir92].

Zdalne wywołanie za pomocą rozgłaszania wymaga, aby przynajmniej jedna procedura została wykonana. Jest to pewnym ograniczeniem, gdyż czasami fakt ten nie ma żadnego znaczenia.

Za pewną niedogodność można uznać konieczność samodzielnego programowania filtrów do konwersji danych, zwłaszcza w przypadku skomplikowanych struktur. W rzeczywistości użytkownik nie musi robić tego sam. Można skorzystać z programu rpcgen, który nie tylko przygotowuje filtry, ale daje wiele innych możliwości, których jednak nie będziemy tu omawiać.

10.2 Przykłady

10.2.1 Wzajemne wykluczanie

Mechanizm zdalnego wywołania procedury z definicji zapewnia wzajemne wykluczanie procesów wołających podczas wykonywania zdalnej procedury. Niech procesy P wywołują procedurę zdalną SEKCJA_KRYTYCZNA, udostępnianą przez proces obsługujący WYKLUCZANIE:

```
const  N = ?;
process P(i:  1..N);
begin
   while true do begin
     wlasne_sprawy;
     WYKLUCZANIE.SEKCJA_KRYTYCZNA
   end
 end; {P}

process WYKLUCZANIE;
 export  procedure SEKCJA_KRYTYCZNA;
begin
   {kod  procedury}
 end;   {SEKCJA_KRYTYCZNA}
begin
 end; {WYKLUCZANIE}
```

A oto kod w języku C dla procesu obsługującego i procesów wołających. Zdefiniujemy najpierw stale dla numeru programu, wersji i procedury, umieszczając je w pliku services.h

```
#define SEKCJA_KRYT_PROG  0x20000000
#define SEKCJA_KRYT_VERS    1
#define SEKCJA_KRYTYCZNA    1
```

Plik WYKLUCZANIE.c zawiera kod źródłowy procesu obsługującego

```
#include <rpc/rpc.h>
#include "services.h"

char *sekcja_krytyczna  (void);

void main (void)
{
  registerrpc  (SEKCJA_KRYT_PROG,  SEKCJA_KRYT_VERS,
                SEKCJA_KRYTYCZNA,  sekcja_krytyczna,
                xdr_void,  xdr_void);
  svc_run ();
}

char *sekcja_krytyczna  (void)
{
  /* kod procedury */
  return (char  *)NULL;
}
```

Oto plik P.c z kodem źródłowym procesów wołających

```
#include <rpc/rpc.h>
#include "services.h"

void main (void)
```

```
{
  while (TRUE)  {
    /* wlasne sprawy  */
    callrpc  ("nwsl850", SEKCJA_KRYT_PROG,
              SEKCJA_KRYT_VERS,  SEKCJA_KRYTYCZNA, xdr_void,
              NULL, xdr_void, NULL);
  }
}
```

Proces obsługujący jest tu wykonywany przez komputer nwsl850.

Implementacja zdalnego semafora

Semafor, nazwiemy go dalej zdalnym, zaimplementujemy jako zmienną dostępną tylko za pośrednictwem procedur procesu obsługującego. To nam zapewni wzajemne wykluczanie podczas operacji na nim. Wstrzymywanie procesów podczas wykonywania operacji P zrealizujemy w następujący sposób. Dla każdego procesu korzystającego ze zdalnego semafora zadeklarujemy lokalny semafor oraz obsługujący go proces. Jeżeli proces nie może natychmiast wykonać operacji P na zdalnym semaforze, to zostaje wstrzymany na swoim lokalnym semaforze. Gdy na semaforze zdalnym wykona się operacja V, to proces go obsługujący wywola procedurę zdalną procesu obsługującego semafor lokalny jednego z wstrzymanych procesów. Proces obsługujący semafor zdalny musi zatem znać identyfikatory wstrzymanych procesów.

Proces SEMAFOR, obsługujący semafor zdalny, udostępnia procedury P i V. Zmienna s służy dwóm celom: jeśli semafor zdalny jest podniesiony, to s jest jego wartością, jeśli jest opuszczony, to modul s jest liczbą wstrzymanych procesów. Proces wołający procedurę P podaje jako jej parametr swój identyfikator. Jeżeli wartość semafora jest dodatnia, to procedura kończy się przyjmując wartość false. W przeciwnym razie identyfikator procesu jest zapamiętywany w kolejce procesów wstrzymanych, a procedura przyjmuje wartość true. W takiej sytuacji proces wołający procedurę SEMAFOR.P usiłuje wykonać operację P na semaforze binarnym SEM_LOKALNY, który inicjalnie musi być opuszczony. Jeżeli podczas wykonywania procedury SEMAFOR.V okaże się, że są procesy wstrzymane, to z kolejki procesów wstrzymanych zostanie usunięty identyfikator najdawniej wstrzymanego procesu i wywołana procedura zdalna ZWOLNIJ procesu ZAWIADOWCA, związanego ze zwalnianym procesem. W tej procedurze zostaje podniesiony semafor SEM_LOKALNY i w rezultacie następuje wznowienie procesu wstrzymanego pod tym semaforem. Oto zapisany w abstrakcyjnej notacji kod procesu obsługującego semafor zdalny:

```
process SEMAFOR;
var s:  integer;        {semafor zdalny}

procedure wstaw(i: integer);
begin ...               {dołączenie do listy numeru 'i'}
end;                    {wstrzymanego procesu wołającego}

function weź: integer;
begin
  weź := ...           {pobranie pierwszego numeru z listy}
end;                    {wstrzymanych procesów wołających}
export function P(i:  integer): boolean;
begin
  s  := s  -  1;
  if s <  0  then       {wstawienie do listy}
    wstaw(i);
  P  := s  <  0
```

```
end;

export procedure V;
begin
  if s <  0  then ZAWIADOWCA(wez).ZWOLNIJ;
  s  := s  +  1
end;
begin
  s  := 1              {wartość początkowa semafora}
end; {SEMAFOR}
```

A oto procesy korzystające z semafora zdalnego i procesy obsługujące semafory lokalne:

```
const N = ?;
var SEM_LOKALNY: array[l..N] of semaphore := (N*0);

process P(i:  1..N);
begin
  while true  do begin
    wlasne_sprawy;
    if SEMAFOR.P(i)  then P(SEM_LOKALNY[i]);
    sekcj a_krytyczna;
    SEMAFOR.V
  end
end;  {P>

processZAWIADOWCA(i: l..N);
export procedure ZWOLNIJ;
begin
  V(SEM_LOKALNY[i])
end;  <ZWOLNIJ}
begin
end;  {ZAWIADOWCA}
```

W notacji abstrakcyjnej proces jest identyfikowany przez indeks, ale w praktyce identyfikowanie procesu trzeba zrealizować inaczej. Indeks był potrzebny, by wywołać procedurę zdalną odpowiedniego procesu ZAWIADOWCA. W systemie Unix musimy użyć nazwy komputera, wykonującego ten proces. W następującym kodzie w języku C parametrem funkcji realizującej procedurę zdalną P będzie nazwa komputera wykonującego proces wołający tę procedurę. Używamy tu funkcji z przykładu 8.2.1. Plik services.h zawiera numery programów, wersji i procesów:

```
#define SEM_PROG    0x20000000
#define SEM_VERS    1
#define SEM_P       1
#define SEM_V       2
#define ZAW_PROG    0x20000001
#define ZAW_VERS    1
#define ZWOLNIJ     1
```

Oto plik SEMAFOR.c zawierający kod źródłowy procesu obsługującego semafor zdalny:

```
#include <rpc/rpc.h>
#include "services.h"

char *P (char  **);  /*  uwaga na typ parametru - wyjaśnienie poniżej */
char *V (void);
```

```
void wstaw  (char *nazwa)
{     /* dołączenie do listy nazwy komputera, na którym czeka proces wołający */
}

char *weź (void)
{
  char nazwa[80];
                    /*  usunięcie z listy pierwszej nazwy */
  return nazwa;     /*  i przekazanie jako wyniku */
}

int s = 1;          /*  semafor zdalny, wartość pocz. 1 */

void main (void)
{
                    /*  zarejestrowanie obu procedur */
  registerrpc  (SEM_PROG, SEM_VERS, SEM_P,
               P, xdr_wrapstring,  xdr_bool);
  registerrpc  (SEM_PROG,  SEM_VERS,  SEM_V,
               V,  xdr_void,  xdr_void);
  svc_run  ();        /*  czekanie na wywołanie */
}

char *P  (char  **nazwa)
{
  static bool_t czeka;
  s-;
  if (czeka =  s<0)
     wstaw (*nazwa);
  return (char *)&czeka;
}

char *V  (void)
{
  if (s < 0)
     callrpc (wez,  ZAW_PROG,  ZAW_VERS,  ZWOLNIJ,
                xdr_void, NULL,  xdr_void, NULL);
  s++;
  return (char *)NULL;
}
```

Jeżeli parametrem procedury zdalnej jest zmienna typu char *, to parametr funkcji odpowiadającej tej procedurze musi być typu char **. Wynika to z zasady, że parametr takiej funkcji musi być wskaźnikiem do obiektu, który sam jest wskaźnikiem. Oto plik P.c z kodem źródłowym procesów wołających:

```
#include <rpc/rpc.h>
#include "services.h"

#define SEMAFOR "nwsl850" /*  nazwa komputera wykonującego proces SEMAFOR */

/* funkcje z przykładu  8.2.1  - plik semafor.c */
extern void deklaracja  (void);
extern void P  (void);

void main  (void)
{
  bool_t czeka;
  char nazwa[80];
```

```
    deklaracja ();
    gethostname(nazwa,80);     /* pobranie nazwy komputera */
    while (1) {
      /* własne sprawy */
      callrpc (SEMAFOR, SEM_PROG, SEM_VERS, SEM_P,
               xdr_wrapstring, &nazwa, xdr_bool, &czeka);
      if (czeka)
         P ();
      /* sekcja krytyczna */
      callrpc (SEMAFOR, SEM_PROG, SEM_VERS, SEM_V,
               xdr_void, NULL, xdr_void, NULL);
    }
  }
```

Oto plik ZAWIADOWCA. c zawierający kod źródłowy procesów obsługujących semafory lokalne:

```
#include <rpc/rpc.h>
#include "services.h"

/* funkcje z przykładu 8.2.1 - plik semafor.c */
extern void deklaracja (void);
extern void inicjacja (int);
extern void V (void);

char *zwolnij (void);

void main (void)
{
  deklaracja ();
  inicjacja (0);
  registerrpc (ZAW_PROG, ZAW_VERS, ZWOLNIJ,
               zwolnij, xdr_void, xdr_void);
  svc_run ();
}

char *zwolnij (void)
{
  V ();
  return (char *)NULL;
}
```

Przekazywanie uprawnienia

Jakkolwiek koncepcja przekazywania żetonu uprawniającego proces do wejścia do sekcji krytycznej jest bardzo prosta, to jej realizacja za pomocą zdalnego wywołania procedur jest dosyć złożona. Nie można zastosować tu rozwiązania takiego jak w CSP (przykład 5.2.1), gdyż prowadziłoby ono do blokady. Wynika to ze sposobu realizacji zdalnego wywołania procedur. Gdyby jeden z SEKRETARZy rozpoczął przekazywanie uprawnienia do następnego SEKRETARZa, ten do jeszcze następnego itd., to ostatni z nich usiłowałby przekazać uprawnienie temu, który rozpoczął cały proces, ale on byłby właśnie zajęty przekazywaniem uprawnienia.

Aby uniknąć blokady, w procesie SEKRETARZ pozostawimy tylko przyjmowanie uprawnienia, natomiast wysyłaniem go będzie zajmował się odrębny proces POSŁANIEC. Lokalne semafory WEJŚCIE i WYJŚCIE służą do synchronizacji procesów PR, SEKRETARZ i POSŁANIEC. Oto kod tych procesów w notacji abstrakcyjnej:

```
const N  = ?;
var WEJŚCIE: array[l..N]  of  semaphore =  (N*0);
    WYJŚCIE: array[l..N]  of  semaphore =  (1,  (N-l)*0);

process  PR(i:  1..N);
begin
   while  true do begin
    P(WEJSCIE[i]);
    sekcj a_krytyczna;
    V(WYJSCIE[i]);
    wlasne_sprawy
   end
end;  {PR}

process  SEKRETARZ(i: l..N);
export procedure UPRAWNIENIE;
begin
   if  czekP(WEJSCIE[i]) >  0 then V(WEJSCIE[i])
                              else V(WYJSCIE[i])
end;  {UPRAWNIENIE}
begin
end;  {SEKRETARZ}

processPOSLANIEC(i: l..N);
begin
   while true do begin
    P(WYJSCIE[i]);
    SEKRETARZ(i mod N  +  l).UPRAWNIENIE
  end
end;   {POSLANIEC}
```

Algorytm Ricarta i Agrawali

W podanym tu rozwiązaniu wykorzystamy rozgłaszanie do wywołania procedury CHCĘ (por. 5.2.1), czyli do zawiadamiania POMOCNIKów o chęci wejścia do sekcji krytycznej. Proces, który chce wejść do sekcji krytycznej, musi jednak najpierw powiadomić swojego POMOCNIKa. W tym celu wywołuje procedurę ZAJMUJĘ. Rozgłaszanie powoduje, że procedura CHCĘ jest wywołana również w procesie POMOCNIK procesu rozgłaszającego. To wywołanie jest ignorowane. Do przekazania sygnału od POMOCNIKa informującego o tym, że proces może wejść do sekcji krytycznej, zastosujemy semafor JUŻ.

Rozgłaszania użyjemy również do wywołania procedury PROSZĘ w procesie, który zwolnił sekcję krytyczną, czyli do wyrażania zgody na ewentualne wejście innych procesów do sekcji krytycznej. Procedura ta jest wywołana w każdym procesie POMOCNIK, niezależnie od tego, czy odpowiadający mu proces PR czeka na wejście do sekcji krytycznej. Aby POMOCNIK nieopatrznie nie podniósł semafora JUŻ, musimy w procedurze PROSZĘ sprawdzać, czy proces chce wejść do sekcji krytycznej. Oto kod w notacji abstrakcyjnej:

```
const  N =  ?;
var JUŻ: array[l..N] of  semaphore  := (N*0);

process PR(i:  1..N);
var t:  integer;
begin
  while true  do begin
    wlasne_sprawy;
    t  := czas_biezacy;
    POMOCNIK(i).ZAJMUJE(t);
    after POMOCNIK(*).CHCE(i, t) do
```

```
    until true;              {nie interesują nas odpowiedzi}
    P(JUZ[i]);
    sekcj a_krytyczna;
    POMOCNIK(i).ZWALNIAM;
    after POMOCNIK(*).PROSZE do
    until true               {nie interesują nas odpowiedzi}
  end
end;   {PR}
processPOMOCNIK(i: l..N);
var  licz,  mójt: integer;
     samchce: boolean;

export procedure ZAJMUJE(t:  integer);
begin
   samchce  := true;
   mójt :=  t;
   licz :=  0
end;   {ZAJMUJE}

export procedure CHCE(j, t:  integer);
begin
   if  (i <> j) and (not samchce or (mójt > t) or
                  (mójt = t)  and (i > j)) then
      POMOCNIK(j).PROSZĘ
end;   {CHCE}

export procedure ZWALNIAM;
begin
   samchce  := false
end;   {ZWALNIAM}

export procedure PROSZĘ;
begin
   if  samchce then            {czeka na zezwolenia}
   if  licz  = N - 2 then
     V(JUZ[i])              {są od wszystkich}
   else licz := licz + 1
end;   {PROSZE}

begin
   samchce  := false
end;   {POMOCNIK}
```

Korzyści z użycia rozgłaszania do wywołania procedury CHCĘ są bezsporne, gdyż procedura ta musi być wywołana w N - 1 procesach POMOCNIK. Natomiast zastosowanie rozgłaszania w celu wywołania procedury PROSZĘ jest dyskusyjne. Jeżeli proces nie czeka na wejście do sekcji krytycznej, to procedura ta jest wywołana niepotrzebnie. Rozgłaszanie warto zastosować, gdy zwykle wiele procesów jednocześnie czeka na wejście do sekcji krytycznej. W przeciwnym razie lepiej skorzystać ze zwykłego wywołania procedury PROSZĘ tylko w czekających procesach (por. 5.2.1).

Podamy teraz kod w języku C. Zawartość pliku services.h jest następująca:

```
#define POMOCNIK   0x20000000
#define ZAJMUJĘ    1
#define CHCĘ       2
#define ZWALNIAM   3
#define PROSZĘ     4
```

Oto plik P.c z kodem źródłowym procesów PR:

```
   #include <rpc/rpc.h>
   #include "services.h"

   #define N 10        /* zakładamy, że jest 10 procesów */
   /* funkcje z przykładu 8.2.1 - plik semafor.c */
   extern void deklaracja (void);
   extern void P (void);

   extern int czas_biezacy (void);

   typedef struct {
     int t;
     char nazwa[80];
   } st;

   int xdr_s (XDR *xdrsp, st *sp)
   {
     return (xdr_int  (xdrsp, &sp->t) &&
             xdr_wrapstring (xdrsp, &sp->nazwa));
   }

   int po_rozgloszeniu (char *res,  struct sockaddr_in *addr)
   {
     return TRUE;
   }

   void main (void)
   {
     deklaracja ();
     gethostname(s.nazwa, 80);  /*  pobranie nazwy komputera*/
     while (TRUE) {
       /* własne sprawy */
     s.t = czas_biezacy  ();
     callrpc  (s.nazwa, POMOCNIK, 1, ZAJMUJĘ,
             xdr_int, &s, xdr_void, NULL);
     clnt_broadcast  (POMOCNIK, 1, CHCĘ,
             xdr_s, &s, xdr_void, NULL, po_rozgloszeniu);
     P  0;
     /* sekcja krytyczna */
     callrpc  (s.nazwa, POMOCNIK, 1, ZWALNIAM,
             xdr_void, NULL, xdr_void, NULL);
     clnt_broadcast  (POMOCNIK, 1, PROSZĘ, xdr_void, NULL,
                   xdr_void, NULL, po_rozgloszeniu);
  }
}
```

Oto plik POMOCNIK.c zawierający kod źródłowy POMOCNIKów

```
#include <rpc/rpc.h>
#include "services.h"

#define N 10        /* zakładamy,  że jest 10 procesów */

/* funkcje z przykładu 8.2.1 - plik semafor.c */
extern void deklaracja (void);
extern void inicjacja (int);
extern void V (void);

int licz;
```

```
int mójt;
bool_t samchce = FALSE;
char nazwa[80];
typedef struct {
  int t;
  char nazwa[80];
} st;

int xdr_s (XDR *xdrsp, st *sp)
{
  return (xdr_int (xdrsp, &sp->t)  &ft
          xdr_wrapstring (xdrsp, &sp->nazwa));
}

char *zajmuję  (int *);
char *chcę     (st s *);
char *zwalniam (void);
char *proszę   (void);

void main (void)
{
  gethostname (nazwa, 80);   /* pobranie nazwy komputera */
  deklaracja ();
  inicjacja (0);
  registerrpc (POMOCNIK, 1, ZAJMUJĘ,
               zajmuję, xdr_int, xdr_void);
  registerrpc (POMOCNIK, 1, CHCĘ,
               chcę, xdr_s, xdr_void);
  registerrpc (POMOCNIK, 1, ZWALNIAM,
               zwalniam, xdr_void, xdr_void);
  registerrpc (POMOCNIK, 1, PROSZĘ,
               proszę,  xdr_void, xdr_void);
  svc_run ();
}

char *zajmuję (int *t)
{
  samchce = TRUE;
  mójt = *t;
  licz = 0;
  return (char *)NULL;
}

char *chcę (st *s)
{
  if ((strcmp(nazwa,s->nazwa) != 0)    /* różne nazwy */
      && (!samchce || (mójt > s->t) ||
          (mójt == s->t) && (strcmp(nazwa,s->nazwa) > 0)))
      callrpc (s->nazwa, POMOCNIK, 1, PROSZĘ,
               xdr_void, NULL, xdr_void, NULL);
  return (char *)NULL;
}

char *zwalniam (void)
{
  samchce = FALSE;
  return (char *)NULL;
}
   char *proszę  (void)
   {
      if (samchce)
```

```
        if (licz  ==  N - 2)
          V();
        else
          licz++;
      return (char *)NULL;
  }
```

10.3 Zadania

10.3.1 Problem podziału

Rozwiąż zadanie 5.3.3 stosując zdalne wywołanie procedur w systemie Unix.

10.3.2 Korygowanie logicznych zegarów

Rozwiąż problem korygowania logicznych zegarów N procesów, przedstawiony w zadaniu 5.3.5, z założeniem, że każdy proces przekazuje swój czas co M obrotów pętli do wszystkich innych procesów. Korygowanie zegarów tą metodą odbywa się bardziej równomiernie niż w zadaniu 5.3.5, gdyż unikamy tu czynnika przypadkowości.

10.3.3 Głosowanie

Rozwiąż zadanie 5.3.6 z założeniem, że proces przekazuje swoje głosy do wyróżnionego procesu-arbitra, który oblicza wyniki głosowania i zwraca je procesom głosującym. Proces ten zastępuje proces SIEĆ i wszystkie procesy LICZ.

10.3.4 Komunikacja przez pośrednika

Rozwiąż zadanie 5.3.7 stosując zdalne wywołanie procedur w systemie Unix.

10.3.5 Powielanie plików

Rozwiąż problem 5.3.2 w wersji umożliwiajacej zagłodzenie pisarzy i w wersji poprawnej korzystając z przekazywania uprawnienia. Rolę uprawnienia powinien pełnić w obu przypadkach tylko jeden komunikat, zawierający informację o stanie systemu.

10.3.6 Powielanie plików - dostęp większościowy

Opisana w zadaniu 5.3.2 metoda korzystania z powielonego pliku wymaga, aby operacja pisania odbywała się jednocześnie na wszystkich kopiach. W przypadku intensywnego pisania metoda ta jest nieefektywna. W metodzie większościowej do wykonania operacji pisania wystarczy wyłączny dostęp do większości, a więc [N/2] + 1 kopii (liczba wszystkich kopii wynosi N). Ponieważ w takim przypadku nie wszystkie kopie będą zawierały te same informacje, każda z kopii musi być ostemplowana czasem ostatniej modyfikacji;

zakłada się, że logiczne zegary poszczególnych procesów są zsynchronizowane. Operacja czytania wymaga dostępu także do większości, czyli [N/2] + 1 kopii, przy czym spośród odczytanych informacji wybiera się zawsze tę z najpóźniejszym stemplem. Dzięki takiej organizacji mamy gwarancję, że nigdy dwie operacje pisania nie wykonają się jednocześnie oraz że zawsze przynajmniej jedna z odczytywanych kopii będzie aktualna.

Załóżmy, że każda kopia, jest obsługiwana przez proces KOPIA, udostępniający dwie procedury zdalne:

```
procedure PISZ(n:integer; r:rekord; t:integer) i
procedure CZYTAJ(n:integer;  var r:rekord;  var t:integer).
```

Znaczenie parametrów jest następujące: n jest numerem zapisywanego lub odczytywanego rekordu, r - odczytywaną lub zapisywaną wartością, a t czasem wykonania operacji.

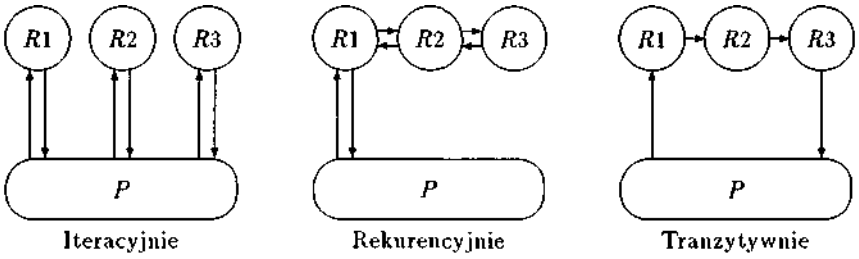
Używając zdalnego wywołania procedur w systemie Unix napisz procedury:

```
pisz(n:   integer;  r:  rekord) i
czytaj(n:  integer;  var r:   rekord),
```

odpowiednio zapisujące i odczytujące określony rekord.

10.3.7 Wyszukiwanie adresów

Każdy obiekt (np. plik, procedura zdalna) w systemie rozproszonym ma globalną nazwę i adres, będący identyfikatorem komputera, na którym ten obiekt się znajduje. Obiekty te są identyfikowane przez nazwy, adresy bowiem mogą się zmieniać, gdy obiekt migruje, lub mogą nie być jednoznaczne, gdy obiekt zostanie powielony. Jedną ze standardowych usług w systemie rozproszonym jest określanie adresu na podstawie podanej nazwy (usługa tajest nazywana name resolution, świadczący ją proces natomiast name server). Proces obsługujący wyszukiwanie adresów utrzymuje bazę danych, w której przechowuje nazwy z odpowiadającymi im adresami. W systemie rozproszonym może być wiele współpracujących procesów wyszukujących nazwy. Baza danych o obiektach jest wówczas rozproszona między te procesy. Każdy proces oprócz swojego fragmentu bazy danych utrzymuje informacje, do którego procesu kierować zapytania o adresy dla nazw, których nie ma w swojej bazie. Wskazany proces może mieć w swojej bazie wskazany adres. Na przykład proces R1 zajmuje się nazwami zaczynającymi się od liter A-D, a zapytania o pozostałe nazwy kieruje do procesu R2, który jednak utrzymuje bazę danych dla nazw zaczynających się tylko od liter E-K, a dla nazw na L-Z zapytania kieruje do procesu R3. W związku z tym stosuje się kilka metod wyszukiwana adresów [Gosc91] (rys. 10.2):



Rys. 10.2. Metody wyznaczania adresów

- Metoda iteracyjna — proces żądający wyszukania adresu sekwencyjnie wywołuje procedury zdalne kolejnych procesów obsługujących, aż adres zostanie znaleziony albo

otrzyma odpowiedź negatywną. Odmianą tej metody jest wywołanie zdalnych procedur we wszystkich procesach obsługujących jednocześnie przez rozgłaszanie.

- Metoda rekurencyjna — proces żądający wyszukania adresu wywołuje procedurę zdalną jednego z procesów obsługujących i czeka na adres albo na informację, że nie ma adresu odpowiadającego podanej nazwie. Proces obsługujący sprawdza, czy nazwa go dotyczy. Jeśli tak, to szuka nazwy w swojej bazie danych. W przeciwnym razie postępuje jak proces żądający wyszukania adresu, w stosunku do innego procesu obsługującego, którego ta nazwa może dotyczyć. W tej metodzie zajętych może być jednocześnie wiele procesów obsługujących, przy czym tylko jeden z nich w danej chwili przeszukuje swoją bazę danych, a inne czekają bezczynnie.
- Metoda tranzytywna — proces żądający wyszukania adresu przekazuje nazwę jednemu z procesów obsługujących i czeka na adres albo informację, że nie ma adresu odpowiadającego podanej nazwie. Proces obsługujący sprawdza, czy nazwa go dotyczy i jeśli tak, to szuka jej w swojej bazie danych. Wynik tego przeszukiwania przekazuje procesowi żądającemu wyszukania adresu. W przeciwnym razie przekazuje nazwę procesowi obsługującemu, którego ta nazwa może dotyczyć. W tej metodzie w każdej chwili zajęty jest tylko jeden proces obsługujący.

Napisz proces żądający wyszukania adresu i procesy obsługujące dla każdej z podanych metod. Można skorzystać z funkcji logicznej znaleziony(n: nazwa; var a:adres): boolean, która dla nazwy n wyszukuje adres a w lokalnej bazie danych i przyjmuje wartość true, jeśli adres został znaleziony, a false w przeciwnym razie, oraz z funkcji dotyczy(n: nazwa). Funkcja ta wywołana w procesie obsługującym wskazuje indeks procesu obsługującego, którego ta nazwa może dotyczyć.

10.4 Rozwiązania

Rozwiązania podajemy używając abstrakcyjnej notacji. Czytelnika zachęcamy do zapisania ich również w języku C, z użyciem funkcji systemu Unix.

10.4.1 Problem podziału

Rozwiązanie to oparto bezpośrednio na rozwiązaniu 5.4.3. Proces TPROC będzie procesem obsługującym, a proces SPROC - procesem wołającym.

```
const S = ?;
      T = ?'

process SPROC;
var ZBIÓRS: array[l..S] of  integer;
    m,  n:  integer;
    dalej :  boolean;
begin
  dalej :=  true;
  while dalej do begin
    m  := MaxElem(ZBIORS);
    n  := TPROC.ZAMIEŃ(m);
    if  m 0 n then ZBIÓRS := ZBIÓRS - {m} +  {n}
    else dalej  := false
  end
end;  {SPROC}
```

```
process TPROC;
var ZBIÓRT: array[l..T] of  integer;
export  function ZAMIEN(m: integer): integer;
var n:  integer;
begin
  n  :=  MinElem(ZBIORT);
  if m  <> n then ZBIÓRT :=  ZBIÓRT  + {m} - {n};
  ZAMIEŃ  := n
end;  {ZAMIEŃ}

begin
end;  {TPROC}
```

Po zakończeniu podziału proces obsługujący będzie nadal oczekiwał na wywołanie procedury ZAMIEŃ.

10.4.2 Korygowanie logicznych zegarów

Wysyłanie czasu do wszystkich procesów najłatwiej zrealizować za pomocą rozgłaszania. Dzięki temu zmniejszamy liczbę komunikatów przesyłanych w sieci.

```
const N = ?;
      M = ?;

process P(i: 1..N);
var z, c, t: integer;
begin
  c := 0;
  z := i;
  while true do begin
    wlasne_sprawy(i);
    c := c + 1;
    if c = z then begin
      t := KONTROLER(i).DAJCZAS;
      if t > c then c  :=  t +  1;
      z := c + M;
      after KONTROLER(*).WEZCZAS(c) do
      until true           {nie interesują nas odpowiedzi}
    end
  end
end;  {P}

processKONTROLER(i: l..N);
var c: integer;

export function DAJCZAS:  integer;
begin
  DAJCZAS := c
end;  {DAJCZAS}
export procedure  WEZCZAS(t:  integer);
begin
   if  t > c then c := t
end;   {WEZCZAS>

begin
   c  := 0
end;   {KONTROLER}
```

Jest to przykład, w którym proces rozgłaszający w ogóle nie jest zainteresowany, czy rozsyłana informacja została odebrana przez jakikolwiek proces. Jednak realizacja zdalnego wywołania procednr w systemie Unix za pomocą rozgłaszania wymaga czekania, aż zakończy się przynajmniej jedna ze zdalnych procednr wywołanych przez rozgłaszanie. Mimo to czas tracony na korygowanie zegarów jest mniejszy, niż gdybyśmy zastosowali metodę podaną w zadaniu 5.3.5.

10.4.3 Głosowanie

Po przekazanin swoich głosów arbitrowi proces głosujący czeka na wyniki głosowania. Arbiter jednak nie może przekazać wyników bezpośrednio procesowi głosującemu. Przekazuje je do POMOCNIKów procesów głosujących za pomocą rozgłaszania. Następnie POMOCNIK przekazuje przez kanał wyniki do procesu głosującego. Wyniki głosowania są przekazywane od arbitra w formie komunikatu, zawierającego dwa pola. Pierwsze z nich niesie informację, czy głosowanie już się zakończyło, drugie - numer wybranego procesu, gdy głosowanie zakończyło się, albo immer procesu, który odpadł w tej turze.

```
   const N = ?;
   type  komunikat =  record
                       koniec: boolean;
                       numer: integer
                      end;
   var K: channel;

   process P(i:  1..N);
   var t: array [l..N]  of integer;
       k: komunikat;
       p, j: integer;
   begin
     for j  :=  1  to N do t[j] :=  1;
     repeat
       ARBITER.LICZ(glosuj(t));
    wez(K, i, p, k);
    if not k.koniec then t[k.numer] := 0
  until k.koniec
end;  {P>

processPOMOCNIK(i: l..N);
export procedure WEZ(k: komunikat);
begin
  wloz(K, i, k)
end;  {WEŹ}
begin
end;  {POMOCNIK}

process ARBITER;
var t: array [l..N] of integer;
    k: komunikat;
    i, j, ile, min: integer;

export procedure LICZ(n: integer);
begin
  if ile = 1 then
    for j := 1 to N do t[j] :=0;
  t[n] := t[n] + 1;
  ile := ile mod N + 1;
```

```
  if ile = 1 then begin   {koniec tury}
    min := 1;             {szukanie niezerowego minimum}
    while t[min] = 0 do
      min := min + 1;
    for j := min + 1 to N do
      if (0 < w[j]) and (w[j]  < w[min])
      then min := j;
    i := 0;
    k.koniec := w[min] = N;
    k.numer := min;
    afterPOMOCNIK(*).WEZ(k) do
    until true            {nie interesują nas odpowiedzi}
  end
end;  {LICZ}

begin
  ile := 1
end;  {ARBITER}
```

Obecność arbitra powoduje, że przedstawione rozwiązanie w gruncie rzeczy jest scentralizowane. Jego zaletą jest fakt odciążenia procesów głosujących od liczenia głosów. Obliczanie glosów jest wykonywane w każdej turze tylko raz przez arbitra. Poza tym w sieci jest przesyłana mniejsza liczba komunikatów.

10.4.4 Komunikacja przez pośrednika

Idea tego rozwiązania jest podobna, jak w rozdz. 5. Proces POŚREDNIK jest jednocześnie procesem obsługującym oraz wołającym procedury zdalne procesów obsługujących POMOCNIK. Problemem jest wstrzymywanie procesów P zgłaszających się po komunikaty, gdy ich dla nich nie ma. Zastosowaliśmy tu następujące rozwiązanie. Proces P może zawsze zgłosić się po komunikat. Jeżeli POŚREDNIK ma komunikat przeznaczony dla tego procesu, to mu go daje. W przeciwnym razie proces czeka na dostarczenie przez kanał komunikatu od POŚREDNIKa. Jest to zawsze kanał o numerze 1, podkanal natomiast jest identyfikowany indeksem procesu. Dzięki temu komunikat trafi zawsze do właściwego procesu.

Zmienna ile[i] oznacza w tym rozwiazaniu liczbę komunikatów, ale jeśli proces P(i) czeka na komunikat, to ile[i] = -1.

```
const N = ?;
      M = ?;
var K:  channel;

process P(i: 1..N);
var k:  komunikat;
    j,  t:  integer;
begin
  while true do  begin
     if not POSREDNIK.CHCE(k,  i)  then wez(K,  i, t, k);
       przetwórz(k);
     losuj(j);
     POSREDNIK.PROSZE(k, j);
     wlasne_sprawy
  end
end;   {P}

processPOMOCNIK(i:  l..N);
```

```
   export procedure PROSZE(k: komunikat);
   begin
     wloz(K,  i, k)
   end;   {PROSZĘ}
begin
end;  {POMOCNIK}

process POŚREDNIK;
var ile: array[l..N] of integer;
    kom: array[l..M] of komunikat;
    adr: array[l..M] of integer;
    i, n: integer;

export function CHCE(var k: komunikat; j: integer):boolean;
begin
  CHCĘ := ile[j] > 0;
  if ile[j] > 0 then begin
    i := 1;
    while adr[i] <> j then i := i+l;
    k := kom[i] ;
    adr[i] := 0
  end;
  ile[j]  := ile[j]-l
end;  {CHCĘ}

export procedure PROSZE(k: komunikat; n:  integer);
begin
  if ile[n] = -1 then
    POMOCNIK(n).PROSZE(k)
  else begin
    i := 1;
    while adr[i] > 0 do i := i+l;
    adr[i] := n;
    kom[i] := k
  end;
  ile[n]  := ile[n] + l
end;  {PROSZĘ}

begin
  for i := 1 to N do ile[i] := 0;
  for i := 1 to M do begin
    losuj(adr[i]);
    ile[adr[i]] := ile[adr[i]]+l;
    kom[i] := nowy_komunikat
  end
end;  {POŚREDNIK}
```

10.4.5 Powielanie plików

Rozwiązanie z możliwością zagłodzenia pisarzy

Rozwiązanie jest podobne do rozwiązania 5.4.2. Różnice wynikają z konieczności wprowadzenia dodatkowych procesów POSŁANIEC, podobnie jak w przykładzie 10.2.1,

```
   const N = ?;
   var MOGĘ: array[l..N] of semaphore  =  (N*0);
       K:  channel;

   process PR(i:  1..N);
   var pisarz: boolean;
```

```
begin
  while true do begin
    pisarz :=  ...;   {określenie rodzaju  procesu}
    SEKRETARZ(i).POZWOL(pisarz);
    P(MOGE[i]);
    if pisarz then pisanie
    else  czytanie;
    SEKRETARZ(i).SKOŃCZYŁEM;
    wlasne_sprawy
  end
end;  {PR}

processSEKRETARZ(i: l..N);
var mam,  chcę, pisarz: boolean;
export procedure POZWOL(czy_pisarz: boolean);
begin
  pisarz  := czy_pisarz;
  chcę := true;
  if not pisarz and mam then  V(MOGE[i])
end;  {POZWÓL}

export procedure SKOŃCZYŁEM;
begin
  chcę := false;
  if pisarz then wloz(K, i, 0)
end;  {SKOŃCZYŁEM}

export procedure UPRAWNIENIE(c: integer);
begin
  if pisarz then
     if chcę and  (c  = 0) then V(MOGE[i])
     else wloz(K,  i,  c)
  else begin {czytelnik}
     if chcę and not  mam then begin
       c := c + 1;
       mam := true;
       V(MOGE[i])
     end;
     if not chcę and  mam then begin
       c := c - 1;
       mam := false
     end;
     wloz(K, i, c)
  end
end;   {UPRAWNIENIE}

begin
  mam := false;
  chcę := false;
  if  i = 1 then wloz(K, 1, 0)
end;   {SEKRETARZ}

processPOSLANIEC(i:  l..N);
var  p:  integer;
begin
  while true do begin
     wez(K, i, p,  c);
     SEKRETARZ(i mod  N + l).UPRAWNIENIE(c)
  end
end;   {POSŁANIEC}
```

Rozwiązanie poprawne

Komunikat opisujący stan systeimi wystarczy rozszerzyć jedynie o liczbę pisarzy, którzy zaczęli czekać podczas czytania. Ponieważ pisarz przetrzymuje komunikat podczas pisania, więc pisarze, którzy w tym czasie zaczęli czekać, nie będą zwiększać tej liczby. Musimy w tym rozwiązaniu zrezygnować z możliwości wielokrotnego czytania podczas jednego obiegu komunikatu. Mogłoby się bowiem zdarzyć, że czytelnik otrzymując komunikat zawsze byłby zajęty czytaniem. W rezultacie nawet jeden czytelnik mógłby zagłodzić pisarzy.

Założenia te oraz połączenie procesów w logiczny pierścień, powoduje że każdy proces w końcu doczeka się na swoją kolej. (To rozwiązanie jest podobne do rozwiązania z przykładu 7.2.3. Tam jednak poprawność wynikała z żywotności mechanizmów Lindy, tu natomiast ze specyficznego mechanizmu synchronizacji, jakim jest przekazywanie komunikatu. Jest to więc bardziej sprawiedliwe rozwiązanie.)

Pisarz zmienia w krążącym komunikacie liczbę czekających pisarzy tylko wtedy, gdy czytelnicy czytają. Zapamiętuje ten fakt na zmiennej czekam, aby w chwili otrzymania komunikatu z zerową liczbą czytających czytelników wiedział, czy ma zmniejszyć liczbę czekających pisarzy.

```
   const N =  ?;
   type komunikat  = record
              c:  integer;      {liczba czytających czytelników}
              p:  integer;      {liczba czekających pisarzy}
        end;

   var MOGĘ:  array[l..N] of semaphore = (N*0);
       K: channel;

   process P(i:  1..N);
   var pisarz: boolean;
   begin
     while true  do begin
       pisarz := ...;         {określenie rodzaju procesu}
       SEKRETARZ(i).POZWOL(pisarz);
       P(MOGE[i]);
       if pisarz then pisanie
       else czytanie;
       SEKRETARZ(i).SKONCZYLEM;
       wlasne_sprawy
     end
   end;   {P}

   processSEKRETARZ(i: l..N);
   var kl: komunikat;       {do przechowywania komunikatu, gdy pisarz pisze}
       chcę,  czekam,  pisarz: boolean;

   export procedure POZWOL(czy_pisarz:  boolean);
   begin
     pisarz := czy_pisarz;
     chcę :=  true
   end;   {POZWÓL}
export procedure SKOŃCZYŁEM;
begin
  chcę := false;
  if pisarz then wloz(K, i, kl)
end;  {SKOŃCZYŁEM}

export procedure UPRAWNIENIE(k:  komunikat);
```

```
begin
  if pisarz then
    if chcę then
      if k.c = 0 then begin
        kl := k;
        if czekam then k.p  := k.p - 1;
        czekam  := false;
        V(MOGE[i])
     end
     else begin
       if not czekam then k.p := k.p +  1;
         czekam := true;
         wloz(K, i, k)
      end
    else wloz(K, i, k)
  else begin {czytelnik}
    if chcę and (k.p = 0) then begin
      k.c := k.c + 1;
      V(MOGE[i])
    end;
    if not chcę and (k.c > 0) then
      k.c := k.c - 1;
    wloz(K, i, k)
  end
end;  {UPRAWNIENIE}

begin
  chcę := false;
  czekam := false;
  if i = 1 then begin
    k.c := 0;
    k.p := 0;
    wloz(K, 1, k)
  end
end;  {SEKRETARZ}
   processPOSLANIEC(i:  l..N);
   var p:  integer;
       k:  komunikat;
   begin
     while true do begin
       wez(K,  i, p, k);
       SEKRETARZ(i mod N +  l).UPRAWNIENIE(k)
     end
   end;   {POSLANIEC}
```

10.4.6 Powielanie plików - dostęp większościowy

Zapisywanie rekordu polega tu na [N/2] + 1 krotnym zdalnym wywołaniu procedury PISZ. Aby zapisywanie dotyczyło różnych kopii, zaczynamy od kopii o losowo wybranym numerze. Odczytanie rekordu natomiast polega na jednokrotnym wywołaniu procedury CZYTAJ z użyciem rozgłaszania. Następnie oczekuje się zakończenia [N/2] + 1 wykonań procedury CZYTAJ. To rozwiązanie jest szybsze od iteracyjnego wywołania procedury CZYTAJ, ale powoduje wykonanie tej procedury we wszystkich N procesach KOPIA.

Podczas operacji pisania dostęp do [N/2] + 1 kopii musi być wyłączny, dlatego w procedurze pisz na początku jest wykonywana procedura

```
poczatek_pisania(i, K: integer).
```

Po jej zakończenhi proces ma zapewniony wyłączny dostęp do K kolejnych kopii, poczynając od tej o numerze i. Proces zwalnia dostęp do kopii wykomijąc procedurę

```
koniec_pisania(i,  K:  integer).
```

Treść procedur poczatek_pisania i koniec_pisania pomijamy. Do ich napisania można użyć jednej z metod opisanych w p. 10.2.1.

```
const  N =  ?;
       K =  N  div 2 + 1; {większość}

procedure  pisz(n:  integer; r:  rekord);
var i,  k:  integer;
begin
  losuj(i);             {procedura  losująca o wartościach całkowitych}
  t :=  biezacy_czas;
  k :=  0;
  poczatek_pisania(i, K);
  repeat
    i  := i mod N + 1;
    KOPIA(i).PISZ(n,  r,  t);
    k  := k +  1
  until k = K;
  koniec_pisania(i,  K)
end;

procedure czytaj(n:  integer; var r: rekord);
var rl: rekord;
    k, t, maxt:  integer;
begin
  maxt  : = 0;
  k := 0;
  after KOPIA(*).CZYTAJ(n,  rl, t) do
    if t > maxt  then begin
      maxt  := t;        {wybór najbardziej}
      r := rl           {aktualnej informacji}
    end;
    k  := k +  1
  until k = K
end;
```

W tym zadaniu założyliśmy, że zegary logiczne poszczególnych procesów są zsynchronizowane. Do ich synchronizacji można zastosować algorytm podany w rozwiązaniu zadania 10.3.2.

10.4.7 Wyszukiwanie adresów

Metoda iteracyjna

Następujące rozwiązanie odnosi się do wersji z rozgłaszaniem.

```
const N = ?;

processRECEPCJONISTA(i: l..N);

export procedure GDZIE(n:  nazwa; var jest: boolean; var a:  adres);
begin
  if dotyczy(n)  =  i  then jest := znaleziony(n, a)
```

```
  else jest := false
end;  {GDZIE}

begin
end;  {RECEPCJONISTA}
process P;
var n: nazwa;
    a: adres;
    i: integer;
    jest: boolean;
begin
  n := ...;    {przypisanie szukanej nazwy}
  i := 0;
  after RECEPCJONISTA(*).GDZIE(n, jest, a) do
    i  := i +  1
  until jest  or  (i  =  N);
  wlasne_sprawy(jest,  a)
end;  {P}
```

Po rozgłoszeniu proces P czeka tak długo, aż otrzyma odpowiedź z informacją o znalezieniu adresu lub otrzyma wszystkie odpowiedzi. W drugim przypadku albo adresu nie znaleziono, albo informacja o adresie nadeszła w ostatniej odpowiedzi.

Metoda rekurencyjna

```
const N = ?;

processRECEPCJONISTA(i:  l..N);
export procedure  GDZIE(n:  nazwa;  var jest: boolean; var a:  adres);
begin
  if dotyczy(n) = i then  jest  :=  znaleziony(n, a)
  else RECEPCJONISTA(dotyczy(n)).GDZIE(n, jest, a)
end;  {GDZIE>
begin
end;  {RECEPCJONISTA}

process P;
var n: nazwa;
    a: adres;
    jest: boolean;
begin
  n := ...;    {przypisanie szukanej nazwy}
  RECEPCJONISTA(l).GDZIE(n, jest, a);
  wlasne_sprawy(jest,  a)
end;  {P}
```

Proces P wywołuje procedurę GDZIE tylko w procesie RECEPCJONISTA o indeksie 1. Jeżeli przedstawiony program zapisalibyśmy w języku C, to zamiast indeksu użylibyśmy nazwy komputera wykonującego ten proces. Proces wołający nie musiałby nic wiedzieć o istnieniu wielu procesów RECEPCJONISTA.

Metoda tranzytywna

Metoda tranzytywna polega na przekazywaniu komunikatów między kolejnymi procesami wyszukującymi. Prezentowane rozwiązanie jest implementacją przekazywania komunikatów za pomocą zdalnego wywołania procedur, podobną do przekazywania uprawnienia.

```
const N =  ?;
```

```
   type  komunikat =  record
                       a:  adres;
                       jest: boolean
                     end;

   var K1:  channel;
       K2:  channel;

   processRECEPCJONISTA(i: l..N);
   export  procedure  GDZIE(n: nazwa);
   begin
     wloz(Kl, i, n)
   end;   {GDZIE}
   begin
   end;   {RECEPCJONISTA}

   process  GONIEC(i:  l..N);
   var n:  nazwa;
       a:  adres;
       p:  integer;
       jest:  boolean;
   begin
     wez(Kl,  i,  p, n);
     if  dotyczy(n) =  i  then begin
       jest := znaleziony(n, a);
       SEKRETARZ.TAM(jest, a)
     end
     else  RECEPCJONISTA(dotyczy(n).GDZIE(n)
   end;  {GONIEC}
process SEKRETARZ;
export procedure TAM(jest:  boolean;  a:  adres);
var k: komunikat;
begin
  k.jest := jest;
  k.a := a;
  wloz(K2,  1, k)
end;  {TAM}
begin
end;  {SEKRETARZ}

process P;
var n: nazwa;
    p: integer;
    k: komunikat;
begin
  n := ...;   {przypisanie  szukanej  nazwy}
  RECEPCJONISTA(1).GDZIE(n);
  wez(K2, 1,  p, k);
  wlasne_sprawy(k.jest,  k.a)
end;  {P}
```

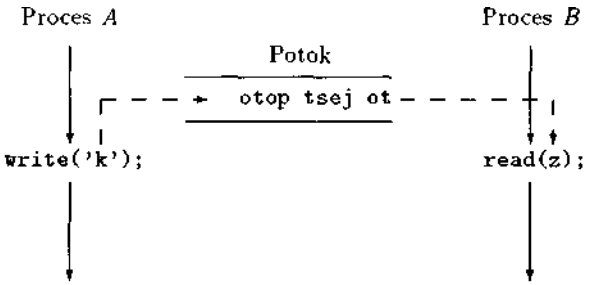
11 Potoki w systemie MS-DOS

11.1 Wprowadzenie

System operacyjny MS-DOS zawiera mechanizm umożliwiający przetwarzanie potokowe (w dość ograniczonym zakresie). Składnia łączenia programów w potok jest taka sama jak w systemie operacyjnym Unix. Zapis

```
pl|p2|p3|...|pn
```

oznacza, że standardowe wyjście programu pl będzie potraktowane jako standardowe wejście programu p2, standardowe wyjście prograimi p2 będzie potraktowane jako standardowe wejście programu p3 itd. Inaczej mówiąc, każdy program (z wyjątkiem pierwszego) będzie czytał to, co wypisze jego poprzednik. Sposób komunikacji przez potok ilustruje rys. 11.1.



Rys. 11.1. Komunikacja przez potok

System MS-DOS nie daje możliwości współbieżnego przetwarzania zadań. Programy pl ,p2,p3, . . . ,pn nie są wykonywane współbieżnie, tak jak w Unixie, ale sekwencyjnie. Najpierw do końca wykonuje się program pl, potem p2, a gdy on się skończy, p3 itd.

Jak więc widać, nie jest to mechanizm ani do programowania współbieżnego, ani tym bardziej do programowania rozproszonego, ponieważ wszystkie programy są wykonywane na jednym tylko komputerze. Wspominamy tutaj o nim co najmniej z dwóch powodów. Po pierwsze, jest to mechanizm zewnętrznie bardzo podobny do potoków w Unixie, warto zatem zwrócić uwagę na istotną różnice w realizacji. Po drugie, nawet jeśli w rzeczywistości wszystko wykonuje się sekwencyjnie, to sposób programowania jest taki sam, jak dla wykonania współbieżnego.

W podanym dalej przykładzie i zadaniach programy napisano w Turbo Pascalu, który jest jednym z najbardziej popularnych języków na komputerach osobistych.

11.2 Przykłady

11.2.1 Producent i konsument

Podane w punkcie 5.4.1 drugie rozwiązanie problemu producenta i konsumenta z rozproszonym buforem ma strukturę potoku. Procesy PRODUCENT, BUFOR i KONSUMENT zapiszemy tu w formie oddzielnych programów, odpowiednio PROD, B i KONS. Program PROD będzie produkował n kolejnych liczb naturalnych wypisując je na standardowe wyjście.

Liczba n będzie parametrem wczytywanym z wejścia. Program B będzie przepisywał swoje standardowe wejście na standardowe wyjście, natomiast program KONS będzie zliczał wczytane liczby i wynik wypisywał na standardowe wyjście.

```
program PROD;
var i,n:  integer;
begin
  readln(n);
  for  i:=l  to n do write(i:4)
end.

program B;
var i:  integer;
begin
  while  not eof do begin
    read(i);
    write(i:4)
  end
end.

program  KONS;
var i,n:  integer;
begin
  n  := 0;
   while not  eof  do begin
     read(i);
     n := n +  1
   end;
   writeln(n)
 end.
```

Po skompilowaniu tych programów i utworzeniu z nich modułów wykonywalnych o takich samych nazwach jak programy, można napisać polecenie

```
PROD|B|KONS
```

Program PROD będzie oczekiwał na podanie liczby, a po jej podaniu i wykonaniu kolejno programów PROD, B i KONS ten ostatni wypisze liczbę wczytanych liczb. Będzie to oczywiście ta sama liczba, którą wczytał program PROD. Ten sam rezultat można uzyskać pisząc:

```
PROD|KONS
```

albo na przykład

```
PROD|B|B|B|B|B|B|B|KONS
```

przy czym w tym drugim przypadku będziemy czekać na wynik odpowiednio dłużej.

W rzeczywistości omawiany przykład ma niewiele wspólnego z problemem synchronizacji producenta i konsumenta. Tutaj producent najpierw wszystko produkuje, a konsument konsumuje dopiero wtedy, gdy wszystkie programy działające jako elementy bufora „przepchną" po kolei swoje wejście na wyjście. Liczba elementów bufora nie ma więc tak naprawdę żadnego znaczenia.

11.3 Zadania

Proponujemy rozwiązanie kilku zadań przedstawionych w rozdz. 5. Układ procesów w tych zadaniach ma strukturę zbliżoną do struktury potoku. Podstawowym problemem jest tu zmiana sposobu komunikacji między procesami tak, aby było to dokładnie przetwarzanie potokowe. Proponujemy, aby sygnały pojawiające się w tych zadaniach, kodować jako znaki końca linii. Znaku końca linii można także użyć do oddzielenia od siebie różnych strumieni danych lub wyróżnienia w strumieniu danych wartości nie kierowanych bezpośrednio do następnego programu w potoku. Ostatnie zadanie jest poświęcone problemowi ośmiu hetmanów, który rozwiązano w rozdz. 7 za pomocą przestrzeni krotek.

11.3.1 Obliczanie histogramu

Rozwiąż zadanie 5.3.4 używając mechanizmu przetwarzania potokowego w systemie MS-DOS. Przyjmij, żejest obliczany histogram dla wartości funkcji sinus w punktach (-10,-9,-8,...,9,10) przy wartościach progowych(-0.8,-0.4,0,0.4,0.8,1.2).

11.3.2 Obliczanie współczynników rozwinięcia dwumianu Newtona (a + b)n

Rozwiąż zadanie 5.3.10 używając mechanizmu przetwarzania potokowego w systemie MS-DOS.

11.3.3 Obliczanie wartości wielomianu

Rozwiąż zadanie 5.3.12 używając mechanizmu przetwarzania potokowego w systemie MS-DOS. Przyjmij, żejest obliczana wartość wielomianu x2 — '2x + '3 w punktach (-10,-9,...,9,10).

11.3.4 Sito Eratostenesa

Rozwiąż zadanie 5.3.14 używając mechanizmu przetwarzania potokowego w systemie MS-DOS. Przyjmij, że układ sit ma generować liczby pierwsze z przedziału [2,161]. (Ile sit ma być w tym przypadku?)

11.3.5 Porównywanie ze wzorcem

Rozwiąż zadanie 5.3.17 używając mechanizmu przetwarzania potokowego w systemie MS-DOS. Przyjmij, że wyszukuje się wystąpienia słowa begin w tekście programu pascalowego.

11.3.6 Problem ośmiu hetmanów

Rozwiąż zadanie 7.3.8 używając mechanizmu przetwarzania potokowego w systemie MS-DOS. Proces j-ty w potoku powinien pobierać od procesu poprzedniego rozwiązanie częściowe z j-1 ustawionymi hetmanami i dostawiać hetmana w kolumnie j. Zrealizuj funkcję próbuj wyznaczając zbiór szachowanych pól w kolumnie j zgodnie z zasadą, że hetman na pozycji (i, k), k<j,szachujepola(i,j), (i-(j-k), j)oraz(i+(j-k), j).

11.4 Rozwiązania

11.4.1 Obliczanie histogramu

W tym przypadku wartości progowe nie mogą trafiać bezpośrednio do odpowiednich programów segregujących, lecz muszą być przekazywane kolejno przez te programy. Każdy z nich pierwszą otrzymaną wartość będzie zachowywał dla siebie, a pozostałe przesyłał dalej. Do oddzielenia wartości progowych od wartości, dla których liczymy histogram, użyjemy znaku końca linii. Jako sygnału końca danych także użyjemy znaku końca linii. Każdy program segregujący będzie sam zliczać odpowiadającą mu wartość histogramu. Sygnał końca danych spowoduje „przepchnięcie" tych wartości przez wszystkie następne programy.

```
program SEGO;
var i: integer;
begin
  for  i:=-2 to 3  do  write(i*0.4:5:l);
  writeln;
  for  i:=-10 to 10 do write(sin(i):6:2);
  writeln
end.

program SEG;
var a,b,x: real;  l,s:  integer;
begin
  if not eoln  then read(a);  {wczytanie progu}
  while not eoln  do  begin   {przesłanie reszty}
    read(b);
    write(b:5:1)
  end;
  readln;
  writeln;
  1 := 0;              {licznik elementów mniejszych od progu}
  while not eoln  do  begin   {wczytywanie x-ow}
    read(x);
    if x >= a  then write(x:6:2)  else 1  := 1 + 1
  end;
  readln;
  writeln;
  while not eof do begin    {przesyłanie histogramu}
    read(s); write(s:4)
  end;
  write(l:4)                {przesłanie swojej wartości}
end.
```

Poprawne wykonanie wymaga tu uruchomienia sześciu programów segregujących, tyle bowiem mamy wartości progowych. Po wprowadzeniu polecenia:

```
SEGO|SEG|SEG|SEG|SEG|SEG|SEG
```

otrzymamy odpowiedź w postaci wektora liczb odpowiadających zadanym wartościom progowym:

```
4 4  2 3 4 4
```

Ponieważ wczytanie wartości progowej w programie SEG odbywa się warunkowo, ten sam wynik otrzymamy uruchamiając większą liczbę procesów SEG w potoku (pozostałe będą wówczas jedynie przekazywać końcowy histogram).

11.4.2 Obliczanie współczynników rozwinięcia dwumianu Newtona (a + b)n

W tym przypadku ostateczny wynik otrzymany w każdym programie P musi być „przepychany" przez wszystkie następne programy tak, aby trafił na wyjście ostatniego z nich.

Rozwiązanie uzyskano przez proste przetłumaczenie rozwiązania 5.4.10 na język Pascal. Program P1 wczytuje daną n ze swego standardowego wejścia, po czym uruchamia n „fal". Liczby przekazywane w poszczególnych falach oddzielono od ostatecznych wyników znakiem końca linii. Program P może nie uczestniczyć w obliczeniach, jeśli nie dotrze do niego żadna fala, ale zawsze musi on przekazywać ostateczne wynikł obliczeń. Ostatnią instrukcją programu P jest przekazanie swojego wyniku, można ją jednak wykonać tylko wtedy, gdy program uczestniczył w obliczeniach, dlatego przedtem jest badana wartość zmiennej x.

```
program P1;
var  n:integer;
begin
  readln(n);
  if n>0 then begin
     write(n-l:4);
     while n>l do  begin
       write(l:4);            {uruchamianie fal}
       n := n -  1
     end
  end;
  writeln;                   {wysłanie pierwszego}
  write(l:4)                 {współczynnika}
end.

program P;
var  x,y,n: integer;
begin
  x  := 0;
   if not  eoln then begin
    read(n);                 {wczytanie liczby fal}
    x  :=  1;
    if  n  = 0  then y := 0
    else  begin
      write(n-l:4);          {liczba fal dla sąsiada}
      read(y);
      while n>l  do begin
        x :=  x + y;
        write(x:4);
        n :=  n - 1;
        read(y)
      end
    end
  end;
  readln;
  writeln;
  while not eof  do begin
    read(n);  write(n:4);      {przepychaniewynikow}
  end;
  if x  >  0 then  write(x+y:4)  {własny wynik}
end.
```

Programy uruchamiamy pisząc na przykład polecenie postaci:

```
Pl|P|P|P|P|P|P|P|P
```

przy czym liczba uruchomionych programów P musi być większa niż podawana przez użytkownika dana n.

11.4.3 Obliczanie wartości wielomianu

Zastosowano tu sposób przekazywania współczynników z wersji 1 i sposób przekazywanie argumentów z wersji 2 zadania 5.3.12. Jako wynik uzyskamy ciąg par (argument, wartość).

```
program PO;
var i:  integer;
begin
  writeln(l:4,-2:4,3:4);      {współczynniki}
  for i := -10 to 10 do       {argumenty}
    write(i:4,0:4);
  writeln
end.
program P;
var a,b,y:  integer;  mam, iksy: boolean;
begin
  mam  : = f als e;
  iksy  := false;
  while not eof  do begin
    if eoln then begin       {zmiana  strumienia}
      readln;
      writeln;
      iksy  := not iksy;
      mam  := mam and iksy
    end else begin
      read(b);
      if iksy then begin     {to byl  argument}
        read(y);  write(b:4); {wczytujemy  sumę częściową}
        if mam then write(y*b+a:4) else write(y:4)
      end else {not iksy}
      if mam then write(b:4) else begin
        mam  := true;
        a  := b                {zapamiętujemy współczynnik}
      end
    end
  end
end.
```

Obliczenia wykonujemy dla wielomianu drugiego stopnia, więc aby poprawnie uruchomić te programy, należy napisać polecenie:

```
PO|P|P|P
```

Można także uruchomić większą liczbę programów P, ale będą one jedynie przekazywać wyniki.

11.4.4 Sito Eratostenesa

Programy SITO muszą przekazywać na wyjście zarówno swoją własną liczbę pierwszą, jak i przesiewane liczby. W celu odróżnienia ich od siebie, liczby pierwsze rozpoznane przez sita odzielimy od reszty liczb znakiem końca linii.

```
program SITOO;
var i: integer;
begin
  writeln(2:3);
  for i :=  1 to  80 do  write(2*i+l:4)
end.
```

```
program SITO;
var  i,k:  integer;
begin
  while not eoln do begin
     read(i);              {przekazanie liczb pierwszych}
     write(i:4)           {od poprzednich sit}
  end;
  readln;
  if not eof  then begin
     read(i);              {pobranie swojej liczby pierwszej}
     write(i:4)
  end;
  writeln;
  while not eof do begin {przesiewanie}
     read(k);
     if  k mod  i <> 0 then write(k:4)
  end
end.
```

SITOO generuje liczby nieparzyste z przedziału [3,161], więc zgodnie z rozważaniami z p. 5.4.14 minimalna liczba sit wystarczająca do odsiania liczb niepierwszych wynosi 4. Poprawne polecenie uruchamiające powyższe programy wygląda następująco:

```
SITOO|SITO|SITO|SITO|SITO
```

11.4.5 Porównywanie ze wzorcem

Wyszukuje się tu wszystkie wystąpienia słowa begin w tekście programu U, przy czym spacje na początku linii są także liczone. Liczby, które w rozwiązaniu z p. 5.4.17 byly przesyłane bezpośrednio do procesu Z, tutaj są przesyłane między programami P, ale w celu odróżnienia ich od analizowanego tekstu występują one między dwoma znakami końca linii. Treść programu Z dokładnie odpowiada treści procesu Z z p. 5.4.17.

```
program U;
const N = 5;
var  c:char;
     plik,z:text;
begin
  writeln(N:4);       {wysłanie  długości wzorca}
  writeln('begin');   {wysłanie  wzorca}
  assign(plik,'u.pas');
  reset(plik);        {otworzenie  pliku z tekstem}
while not eof(plik) do begin
  while not eoln(plik) do begin
    read(plik,c);
    write(c)        {przesłanie tekstu}
  end;              {z pominięciem}
  readln(plik)      {znaków końca linii}
end;
close(plik)
end.

program P;
var a,b: char;   i,j,n: integer;
begin
  readln(i);          {wczytanie numeru procesu}
  if i>l then writeln(i-l);
  for j := 1 to i-1  do begin
```

```
    read(b);
    write(b)          {przesłanie elementów wzorca}
  end;
  if i>l then writeln;
  readln(a);          {wczytanie elementu wzorca}
  n := 1;              {licznik znaków}
  while not eof do begin
    while eoln do begin
      readln;
      readln(j);
      writeln;
      writeln(j:5)
    end;
    read(b);          {wczytywanie tekstu}
    if i 0 1 then write(b);
    if b = a then begin
      writeln;
      writeln(n-i+i:5)
    end;
    n := n + 1
  end
end.

program Z;
const N = 5;
var m,j: integer;   licz, widz:  array[O..N-l]  of integer;
   begin
     for j  :=  0  to N-1 do widz[j]  :=  -N;
     while  not eof do begin
       readln;
       readln(m);
       j := m  mod  N;
       if m <> widz[j] then begin
         widz[j]  := m;
         licz[j]  := 1
       end  else  begin
         licz[j]  := licz[j] + 1;
         if licz[j] = N then write(m:5)
       end;
     end;
   end.
```

Ponieważ we wzorcu mamy pięć znaków, w celu poprawnego uruchomienia programów należy wydać polecenie:

```
U|P|P|P|P|Z
```

Jeśli program U przepisano dokładnie, w wyniku powinniśmy otrzymać liczby

```
50 114 240  273
```

11.4.6 Problem ośmiu hetmanów

W rozwiązaniu za pomocą mechanizmu potoków nie ma problemu ze stwierdzeniem, kiedy zakończyły się obliczenia. Proces inicjujący INIT redukuje
się tu do jednej instrukcji wypisania ukladu reprezentującego pustą szachownicę. Algorytm procesu wykonawcy W jest podobny do algorytmu z p. 7.4.8.

```
program  INIT;
begin
```

```
  writeln(l:2,0:2,0:2,0:2,0:2,0:2,0:2,0:2,0:2)
end.

program  W;
var u: array[1..8] of integer;
                        {tablica z układem}
    i,j: integer;
    z: set  of 1..8;     {zbiór  pól szachowanych}
begin
  while  not eof do begin
    readln(j ,u[l] ,u[2] ,u[3] ,u[4] ,u[5] ,u[6] ,u[7] ,u[8]);
    z := [];
    for  i:=l  to j-1 do  {wyznaczanie pól szachowanych}
      z  :=  z  + [u[i]] + [u[i]-(j-i)]  + [u[i] + (j-i)];
    for  i  :=  1 to 8 do  {dostawianie hetmana}
      if not  (i in z) then begin
                        {pole nieszachowane}
        u[j]  := i;      {tu można  dostawić}
        writeln(j + l,u[l] :4,u[2] :2,u[3] :2,
                 u[4] :2,u[5] :2,u[6] :2,u[7] :2,u[8] :2)
      end
  end
end.
```

Wydając polecenie

```
INIT|W|W|W|W|W|W|W|W >wynik
```

na pliku wynik otrzymamy wszystkie 92 rozwiązania. W rzeczywistości tylko 12 jest istotnie różnych. Pozostałe powstają z nich przez obrót lub symetrię (por. [WirtSO]).

Literatura

Wszystkich zainteresowanych problematyką programowania współbieżnego gorąco zachęcamy do przeczytania doskonałego podręcznika Ben-Ariego [BenA89], w którym w bardzo przystępny sposób opisano podstawowe problemy i mechanizmy synchronizacji procesów. Znaczne jego fragmenty zawarl Ben-Ari w kolejnym podręczniku [BenA90] poświęconym przede wszystkim zagadnieniom programowania rozproszonego. Można tam znaleźć krótkie wprowadzenie do programowania w Adzie, occamie i Lindzie, uwagi na temat implementacji tych języków i kilka interesujących algorytmów rozproszonych. Obszerny przegląd mechanizmów programowania współbieżnego zawiera także monografia [IsMa82], która jednak nie obejmuje nowszych pomysłów, takich jak occani czy Linda. Różne aspekty programowania współbieżnego bardzo dostępnie zostały omówione w rozdziale 10 książki Harela [Hare92]. Książka [Rayn88] omawia wiele niebanalnych algorytmów rozproszonych, których implementacja za pomocą różnych omawianych tu mechanizmów może być także ciekawym ćwiczeniem. O metodach dowodzenia poprawności programów współbieżnych i rozproszonych można przeczytać w artykułach [OwGr76] i [Lamp82].

Programowanie współbieżne powstało na gruncie teorii systemów operacyjnych. Wiele przykładów i zadań w tej książce dotyczy zagadnień ściśle związanych z zarządzaniem zasobami komputerowymi. Najlepszym znanym autorom podręcznikiem poświęconym systemom operacyjnym jest [SiPSG91]. Bardziej teoretyczne podejście prezentują książki [BiSh88, MaOO87]. W języku polskim dostępne są także podręczniki [Briii79], [Shaw79] i [MaDo83], ale są one dziś już nieco przestarzałe. Obszerną monografią na temat rozproszonych systemów operacyjnych jest [Gosc91]. Można w niej znaleźć opisy rozproszonych algorytmów elekcji, wykrywania blokady, rozdawania kluczy do szyfrowania informacji, wyszukiwania adresów. Rozproszonym systemom operacyjnym są poświęcone także książki [CoDo88, SlKr87] oraz części książek [SiPG91, Tane92, Baco93]. Przystępne wprowadzenie w tę problematykę znajdzie czytelnik w artykule [Weis93].

Odwołujemy się tu do dwóch konkretnych systemów operacyjnych: Unixa i MS-DOSu. Podstawowe informacje dotyczące tych systemów można np. znaleźć odpowiednio w podręcznikach [Silv90, Roch93] oraz [Demi90]. Szczegółowe informacje na temat systemu Unix zawierają dokumentacje [ATfeT90a, ATfeT90b, Berk86]. O implemetacji tego systemu traktują książki [Bach86, LMKQ89].

Mechanizmy opisane w tej książce są stosowane w różnych językach programowania współbieżnego. Żadnego z tych języków nie omówiliśmy tu jednak w całości. Czytelników zainteresowanych szczegółami odsyłamy do odpowiednich podręczników, raportów i opracowań:

- Ada [Ledg80, LeVe85, Pyle86, Booc87, WaWF87, AtMN88, Barn89, BenA89, HaPe89, Shum89, BenA90]
- C pod Unixem [KeRi87, Silv90]
- Concurrent C [GeRo88]
- Concurrent Euclid [Holt83]
- Concurrent Pascal [Brin75, IsMa83]
- CSP [Hoar78]
- Edip [DuWy90a, DuWy90b]
- Linda [Gele85, AhCG86, CaGL86, ACGK88, CaGe89, BenA90, CzZi93]
- Mesa [LaRe80]
- Modula 2 [Glea92, Terr92, Wirt87]
- Modula 2+ [Rovn86]

- Modula 3 [Nels91]
- occam 2 [Burn88, Gall90, Inmo88, Jone87, PoMa87, BenA90]
- Parallel C [Inmo90, Para91]
- Pascal (jako środowisko dla różnych mechanizmów) np. [IgMM92]
- PascaLC [Kurp87]
- Pascal Plus [WeBu79, BuEW88]

Wprowadzenie w problematykę programowania równoległego i obliczeń równoległych stanowią, książki [Axfo89, LeE192]. W artykule [BaST89] można znaleźć przegląd języków programowania rozproszonego, a w artykule [Andr91] przegląd technik programowania rozproszonego.

Niniejsza książka jest co prawda przeznaczona głównie dla studentów, ale może być także przydatna dla nauczycieli prowadzących lekcje informatyki w szkołach średnich, jako źródło interesujących problemów i ich rozwiązań. Przystępne wprowadzenie w problematykę programowania współbieżnego adresowane do nauczycieli i uczniów szkół średnich stanowi cykl artykułów w czasopiśmie „Komputer w Szkole" [Weis90a, Weis90b, Weis91a, Weis91b, Weis91c, Weis91d].

A oto pełny spis cytowanej literatury:

[AhCG86] S. Ahuja, N. Carriero, D. Gelernter: Linda and friends. IEEE Computer, 1986, 19, 8, 26-34.

[ACGK88] S. Ahuja, N. Carriero, D. Geletnter, V. Krishnaswamy: Matching Language and Hardware for Parallel Goinputation in the Linda Machine. IEEE Trans, on Computers, 1988, 37, 8.

[Andr91a] G. R. Andrews: Paradigms for Process Interaction in Distributed Programs. ACM Computing Surveys, 1991, 23, 1, 49-90.

[Andr91b] G. R. Andrews: Concurrent Programming: Principles and Practice. Redwood City CA, Benjamin/Cummings 1991.

[AtMN88] C. Atkinson, T. Moreton, A. Natah: Ada for distributed systems. Cambridge, Cambridge University Press 1988.

[ATfcT90a]ATfcT, UnixSystem VReleaseA -Programmer's Guide: NetworkingInterfaces. Englewood Cliffs, NJ, Prentice-Hall 1990.

[ATfeT90b] ATfeT, Unix System V Release A - Programmer's Reference Manual. Englewood Cliffs, NJ, Prentice Hall 1990.

[Axfo89] T. Axford: Concurrent programming, fundamental techniques for real-time and parallel software design. Chichester, Wiley 1989.

[Bacli86] M, J. Bach: The Design of the Unix Operating System. Englewood Cliffs, NJ, Prentice Hall 1986.

[Baco93] J. Bacon: Concurrent systems. An integrated approach to operating systems, database and distributed systems. Wokingham, England, Addison-Wesley 1993.

[BaST89] H. E. Bal, J. G. Steiner, A. S. Tanenbaum: Programming languages for distributed systems. ACM Computing Surveys, 1989, 21, 3, 261-322.

[Barii89] J. G. P. Barnes: Programming in Ada. Wyd. 3, Wokingham, England, AddisonWesley 1989.

[BenA89] M. Ben-Ari: Podstawy programowania współbieżnego. Warszawa, WNT 1989.

[BenA90] M. Ben-Ari: Principles of concurrent and distributed programming. Englewood Cliffs, NJ, Prentice-Hall 1990.

[Berk86] Berkeley, Unix Programmer's Reference Manual (PRM), Ą.3 Berkeley Software Distribution. Computer Systems Research Group, Computer Science Division, Univ.

of California, Berkeley, Calif. 1986.
[Bern80] A. J. Bernstein: Output guards and nondeterminism in CSP. ACM Tran$. Program. Lang. Syst., 1980, 2, 2, 234-238.
[BeLe93] A. J. Bernstein, P. M. Lewis: Concurrency inprogrammning anddatabase systems. Boston, Jones and Barlett 1993.
[BiSli88] L. Bic, A. C. Shaw: The. logical design of operating systems. Wyd. 2, Englewood Cliffs, NJ, Prentice-Hall 1988.
[Booc87] G. Booch: Software engineering with Ada. Wyd. 2, Menlo Park, California, Benjamin/Cummings 1987.
[Brin75] P. Brinch Hansen: The programming language - Concurrent Pascal. IEEE Trans, on Soft. Eny., 1975, SE-1, 2, 199-207, także w: Programming methodology, D. Gries (ed.), Berlin, Springer 1978.
[Brin77] P. Brinch Hansen: The architecture of concurrent programs. Englewood Clifs, NJ, Prentice-Hall 1977.
[Brin78] P. Brinch Hansen: Distributed processes: a concurrent programming concept. Communication ofthe ACM, 1978, 21, 11, 934-941.
[Brin79] P. Brinch Hansen: Podstawy systemów operacyjnych. Warszawa, PWN 1979.
[Brin83] P. Brinch Hansen: Programming a Personal Computer. Englewood Cliffs, NJ, Prentice-Hall 1983.
[Brin86] P. Brinch Hansen: The Joyce language report. Computer Science Department, University of Copenhagen, Copenhagen 1986.
[Burii88] A. Burns: Programming in occam 2. Reading, Mass., Addison-Wesley 1988.
[BuEW88] D. Bustard, J. Elder, J. Welsh: Concurrent program structures. Englewood Cliffs, NJ, Prentice-Hall 1988.
[CaHa74] R. H. Campbell, A. N. Habermann: Thespecification ofprocess synchronization by path expression, w: Lecture Notes in ConiputerScience, 16, Springer 1974, 89-102.
[CaGL86] N. Carriero, D. Gelernter, J. Leichter: Distributed data structures"m Linda. 13th ACM Symposinm on Principles of Programming Languages, ACM SIGPLAN and SIGACT, (Wiiliamburg, Virginia, 1986), 236-242.
[CaGe89] N. Carriero, D. Geletnter: How to write parallel programs: A guide to the perplexed. ACM Computing Survays, 1989.
[CoDo88] G. F. Coulouris, J. Dollimore: Distributed systerns: concepts and design. Reading, Mass., Addison-Wesley 1988.
[CoHP71] P. J. Courtois, F. Heymans, D. L. Parnas: Concurrent control with "readers" and "writers". Communication of the ACM, 1971, 14, 10, 667-668.
[CzZi93] G. Czajkowski, K. Zielinski: Linda - środowisko do przetwarzania równoległego i rozproszonego w sieciach stacji roboczych. Informatyka, 1993, 5.
[Demi90] J. Deminet: System operacijjny MS-DOS. Warszawa, WNT 1990.
[Dijk68] E. W. Dijkstra: Cooperating sequential processes, W: Programming Languages, A. Genuys (ed.), London, Academic Press 1968, 43-112.
[Dijk71] E. W. Dijkstra: Hierarchical ordering ofsequential processes. Acta Inf., 1971, 1, 115-138.
[Dijk78] E. W. Dijkstra: Umiejętność programowania. Warszawa, WNT 1978.
[DuWy90a] C. Dubnicki, W. Wyglądała: EDIP project - The Edip Language Report, version 1.0. Sprawozdania Instytutu Informatyki Uniwersytetu Warszawskiego, nr 179, Warszawa 1990.
[DuWy90b] C. Dubnicki, W. Wyglądała: Edip project - EDIP PC User's Guide, EDIP project - EDIP PC Implementation Notes, version 1.0. Sprawozdania Instytutu Informatyki Uniwersytetu Warszawskiego, nr 180, Warszawa 1990.
[Gall90] J. Galletly: Occam 2. London, Pitman Publishing 1990.

[GeRo88] N. Gehaiii, W. I). Roome: The Concurrent Cprogramminglanguage. Summit, NJ, Silicon Press, 1988.
[Gele85] D. Gelernter: Generative communication in Linda, ACM Trans. Program. Lang. and Stjst., 1985, 7, 1, 80-112.
[Glea92] R. Gleaves: Modula 2 dla programujących w Pascalu. Warszawa, WNT 1992.
[Gosc91] A. Gościński: Distributed operating systems, The logical design. Reading, Mass., Addison-Wesley 1991.
[HaPe89] A. N. Haberitiann, D. E. Perry: Ada dla zaawansowanych. Warszawa, WNT 1989.
[Hare92] D. Harel: Rzecz o istocie informatyki. Algorytmika. Warszawa, WNT 1992.
[Hoar74] G. A. R. Hoare: Monitors: an operating system structuring concept. Communications of the ACM, 1974, 17, 10, 549-557.
[Hoar78] C. A. R. Hoare: Communicating sequential processes. Communications of the ACM, 1978, 21 8, 666-677.
[Hoar86] C. A. R. Hoare: Communicating sequential processes. Englewood Cliffs, NJ, Prentice-Hall 1986,
[Holt83] R. C. Holt: Concurrent Euclid, The Unix System and Tunis. Reading, Mass., Addison-Wesley 1983.
[IgMM92] M. Iglewski, J. Madey, S. Matwin: Pascal. Wyd. 5, Warszawa, WNT 1992.
[Inmo88] Ininos Ltd, Occam 2 Reference. Manual. Englewood Cliffs, NJ, Prentice-Hall 1988.
[Inmo90] Iiimos Ltd, Inmos ANSI C toolset user manual. 1990.
[IsMa82] W. Iszkowski, M. Maniecki: Programowanie współbieżne. Warszawa, WNT 1982.
[Jone87] G. Jones: Programming in occam. Englewood Cliffs, NJ, Prentice-Hall 1987.
[KeRi87] B. W. Kernighan, D. M. Ritchie: Język C. Warszawa, WNT 1987.
[Kurp87] S. Kurpiewski: .Jezyk PascaLC, Instytut Informatyki Uniwersytetu Warszawskiego, Warszawa 1987, praca magisterska.
[Lamp78] L. Lamport: Time, clock, and the ordering of events in distributed systems. Communications ofthe ACM, 1978, 21, 7, 558-565.
[Lamp82] L. Lamport: An assertional correctness proof of a distrubuted algorithm. Sci. Comput. Program., 1982, 2, 3, 175-206.
[Lamp77] B. W. Lampson i in.: Report in the programming language Euclid. ACM Sigplan Notices, 1977, 12, 2.
[LaRe80] B. W. Lampson, D. D. Redell: Experience with processes and monitors in Mesa. Communication ofthe ACM, 1980, 23 2.
[Ledg80] H. Ledgard: Ada. An introduction. Razem z: Ada reference manual (July 1980). New York, Springer-Verlag 1980.
[LFJL86] S. J. Leffler, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, C. Torek: An Advanced 4.3BSD Interprocess Communication Tutorial, Unix Programmer's Supplementary Documents. Vol. 1 (PS1), 4.3 Berkeley Software Distribution, Computer Systems Research Group, Computer Science Division, Univ. of California, Berkeley, Calif., 1986.
[LMKQ89] S. J. Leffler, M. I<. McKusik, M. J. Karels, J. S. Quarterman: The Design and Implementation of the A.3BSD Unix Operating System. Reading, Mass., AddisonWesley 1989.
[LeVe85]] D. Le Verrand: Evaluating Ada. New York, Macmillan 1985.
[LeE192] T. G. Lewis, H. El-Rewini: Introduction to parallel computing. Englewood Cliffs, NJ, Prentice-Hall 1992.
[Lipt74] R. .J. Lipton: A comparative study of models of parallel compntation. 15th Symposium on Switching and Automata Theory, 1974, 145-155.
[MaDo83] S. E. Madnick, J. J. Donovan: Systemy operacyjne. Warszawa, PWN 1983.

[MaOO87] M. Maekawa, A. E. Oldehoeft, R. R. Oldehoeft: Operating Systems: Advanced concepts. Meiilo Park, Calif., Benjamin/Cummings 1987.
[Nels91] G. Nelson (ed.): Systems Programming with Modula-3. Englewood ClifFs, NJ, Prentice-Hall 1991.
[OwGr76] S. Owicki, D. Gries: An axiomatic proof technique for parallel programs. Acta Inf. 1976, 6, 319-340.
[Para91] Parallel C User Guide, 3L Ltd. 1991.
[Pati7l] S. S. Patil: Limitation and capabilities of Dijkstra's semaphore primitives for coordination among processes, MIT Project MAC Corporation Structure Group Memo 57, MIT, Cambridge, Mass. 1971.
[Pete81] G. L. Peterson: Myths about the mutual exclusion problem. Information ProcessingLetters, 1981, 12, 3.
[PoMa87] D. Pountain, D. May: A Tutorial to occ.am Programming. Blackwell Scientific Publication 1987.
[Prog91] Z. Czech (ed.): Programowanie współbieżne. wybrane zagadnienia. Politechnika Sląska, skrypty uczelniane, nr 1638, Gliwice 1991.
[Pyle86] I. C. Pyle: Ada. Warszwa, WNT 1986.
[Rayn88] M. Raynal: Distributed algoritms andprotocols. Chichester, Wiley 1988.
[RiAg81] G. Ricart, A. Agrawala: An optimal algorithm for mutual exclusion in computer networks. Communications ofthe ACM, 1981, 24, 1, 9-17.
[Roch93] M. J. R.ochkind: Programowanie w systemie Unix dla zaawansowanych. Warszawa, WNT 1993.
[RoKF92] W. Rosenberg, D. Kenney, G. Fisher: Understanding DCE. O'Reilly fe Ass., 1992.
[Rovn86] P. Rovner: Extending Modula-2 to build large, integrated systems. IEEE Software, 1986, 3 C.
[Sant91] M. Santifaller: TCP/IP and NFS, Internetworking in a Unix environment. Wokingham, England, Addison-Wesley 1991.
[Shaw79] A. C. Shaw: Projektowanie logiczne systemów operacyjnych. Warszawa, WNT 1979.
[Shir92] J. Shirley: Guide to writing DCE applications. O'Reilly fe Ass., 1992.
[Shiim89] K. Shumate: Understanding Ada with abstract data types. Wyd. 2, New York, Wiley 1989.
[SiPG91] A. Silberschatz, J. Peterson, P. Galvin: Operating system concepts. Wyd. 3, Reading, Mass., Addison-Wesley 1991 (tłumaczenie w przygotowaniu).
[Silv90] P. P. Silvester: System operacyjny Unix. Warszawa, WNT 1990.
[SlKr87] M. Sloman, J. Kramer: Distributed systems and computer networks. Englewood Cliffs, NJ, Prentice-Hall 1987.
[Stev90] W. R. Stevens: Unix network programming. Englewood Cliffs, NJ, Prentice-Hall 1990.
[SuiiM87] Sun Microsystems, XDR: External Data Representation Standard, RFC 1014, 1987.
[SunM88] Sun Microsystems, RPC: Remote Procedure Call, Protocol Specification, Version 2, RFC 1057, 1988.
[Tane92] A. S. Tanenbaum: Modern operating systems. Englewood Cliffs, NJ, PrenticeHall 1992.
[Terr92] P. D. Terry: Uczymy się programować na przykładzie Moduli 2. Warszawa, WNT 1992.
[Vaiit72] H. Vantigorgh, A. Van Lamsweerde: On an execution of Dijkstra's semaphore primitives. Information ProcessingLetters, 1971, 1, 181-186.

[WaWF87] D. A. Watt, B. A. Wichmann, W. Findlay: Ada language and methodology, Englewood Cliffs, NJ, Prentice-Hall 1987.
[Weis90a] Z. Weiss: O procesach współbieżnych na lekcji informatyki. Komputer tu Szkole, 1990, 11.
[Weis90b] Z. Weiss: Jak zrealizować wzajemne wykluczanie procesów? Komputer w Szkole, 1990, 12.
[Weis91a] Z. Weiss: O błędnych kołach i martwych punktach. Komputer w Szkole, 1991, 1.
[Weis91b] Z. Weiss: Spotkania a programowanie współbieżne. Komputer w Szkole, 1991, 2.
[Weis91c] Z. Weiss: Ludzie listy piszą. Komputer w Szkole, 1991, 3.
[Weis91d] Z. Weiss: Komputerowe wybory. Komputer w Szkole, 1991, 9.
[Weis93] Z. Weiss: Rozproszone systemy operacyjne. Informatyka, 1993, G.
[WeBu79] J. Welsh, D. W. Bustard: Pascal Plus: another language for modular multiprogramming, Software Practice and Experience, 1979, 9, 11, 947-958.
[WeLi81] J. Welsh, A. Lister: A comparative study of task communication in Ada. Software Practice and Experience, 1981, 11, 257-290.
[Wils91] J. Wilson: Berkeley Unix, A simple and comprehensive guide. Wiley 1991.
[Wirt80] N. Wirtli: Algorytmy + struktury danych = programy. Warszawa, WNT 1980.
[Wirt87] N. Wirth: Modula 2. Warszawa, WNT 1987.

Dziękuję za pomoc w stworzeniu elektronicznej wersji tej książki dla KAOLIN ☺ ; MARO ☺ ; TOMC ☺.

Nie jest to orginał, więc może zawierać błędy. Dlatego zachęcam do nabycia tejże książki w formie drukowanej.

Pzozdrawiam i do .....
.....
.....
..... następnej publikacji ☺

JAROK ☺