CD-ROM: Inprise (Borland) JBuilder 3 Foundation dla Linuxa i inne hity!

3/2000

magazyn

LINUX & UNIX

IPv6 w praktyce

Szyfrowanie poczty

Solaris 8

ImageMagick

JavaScript w praktyce

Rozszerzenia do Apache'a

Inprise JBuilder 3 Foundation

CD ROM

NR 3(19)/2000 MARZEC CENA 17 zł ISSN 1506-1299

Adres kontaktowy: magazyn@tao.com.pl

E-commerce w świecie Unixa na przykładzie..... 8
InterMarket 3D część VI (ostatnia)

Tworzenie dynamicznych dokumentów PDF w TEX-u..... 10
Ryszard Raszplewicz

ImageMagick - jak sobie ułatwić życie..... 15
Radosław Tryc

Produkty

Środowisko operacyjne Solaris 8..... 18
opracował Marcin Koprowski

Serwer aplikacji Lotus Domino 5..... 21
opracował Janusz Wirtz

JBuilder 3 Foundation..... 23

Nowości i.Sell 2.0..... 24
opracował Janusz Wirtz

Programowanie

JavaScript w praktyce - część III..... 26
Wiktor Łukasik

Technologie sieciowe

Apache - wybrane moduły..... 30
Wojciech Bojdoł

Firma TAO Systems Polska, wydawca miesięcznika Linux & UNIX poszukuje osób do testowania i opisywania programów działających w systemie Linux. Zainteresowane osoby prosimy o kontakt z Redaktorem Naczelnym - Marcinem Koprowskim.
tel. 0501-134656, e-mail: marcin.koprowski@tao.com.pl

Adres kontaktowy: magazyn@tao.com.pl

6bone w praktyce 31
Wojciech Bojdoł

Forwarding poczty z powiadomieniem 33
na telefon komórkowy
Tomaz Gliczyński

Modele obliczeń rozproszonych 37
Piotr Koch

Firewall WatchGuard Firebox II 40
Tadeusz Knapik

Usługi katalogowe NDS eDirectory w e-biznesie 44

Linux dla każdego

Porównanie przeglądarek tekstowych 49
Wojciech Bojdoł

PGP - szyfrowanie poczty 49
Wojciech Bojdoł

TkDesk - elastyczne środowisko pracy 50
w systemach Unix
Karol Grela

Zapraszamy na naszą stronę www
www.magazyn.tao.com.pl

Modele obliczeń rozproszonych

Piotr Koch

Język Fortran-K

Fortran-K jest językiem opartym w dużym stopniu na Fortranie-90. Jest to akademicki język programowania, który nie został opracowany na potrzeby rzeczywistych systemów równoległych, ale powstał by wspierać emulator takich systemów. Dzięki wybraniu takiego właśnie języka, wspierającego różne modele obliczeń i systemów równoległych, możemy lepiej przyjrzeć się cechom porównywanych modeli. Przez zastosowanie ujednoliconej składni Fortanu-K nie skupiamy uwagi na różnicach wynikających ze stosowania różnych języków, ale możemy się przyjrzeć specyficznym konstrukcjom istniejącym w poszczególnych modelach. W opisywanych modelach znajdą zastosowanie cztery dialekty tego języka, CSP (model zadań i kanałów), CR (model krystaliczny), MP (model z proceduralnym przesyłaniem komunikatów) i GC (model z komunikacją generowaną).

Model zadań i kanałów CSP (Communicating Sequential Processes)

Idea programowania bazującego na kanałach została rozwinięta przez Hoare'a, w pracy znanej jako Communicating Sequential Processes (CSP), która jest obecnie najbardziej znaną implementacją języka nazwanego Occam. Fundamentalną ideą modelu CSP jest wykonywanie niezależnych procesów, komunikujących się między sobą poprzez wysyłanie komunikatów przez specjalnie oznaczone kanały komunikacyjne. Procesy nie mają nazw, a komunikować mogą się tylko, jeśli są połączone kanałem.

Do tworzenia procesów, języki bazujące na modelu CSP, takie jak na przykład Fortran-K (CSP) dostarczają specjalną konstrukcję *create:*

```
create
  N_A of A(... parametry ...)
  N_B of B(... parametry ...)
.
.
.
  N_Z of Z(... parametry ...)
end create
```

gdzie $N_A, N_B, \ldots, N_Z$ - liczba procesów, kolejno A, B, ... i Z

Wykonywanie tych procesów może odbyć się w dowolnym porządku, a proces macierzysty (powołujący te procesy), może kontynuować swoje wykonywanie, dopiero wówczas, gdy wykonywanie wszystkich procesów dzieci (powołanych przez niego procesów) zostanie przerwane.

Zamiast używać semaforów do kontroli dostępu do zmiennych dzielonych, w modelu CSP potrzebne są procesy komunikujące się przez kanały. Kanały są wbudowanymi typami rekordów, jak ciągi znakowe czy urządzenia, i deklaruje się je zazwyczaj w następujący sposób:

```
type (chan) :: c, d, y(10)
.
.
.
type handle
type (chan), auto:: c
real              :: buffer(128)
end type
```

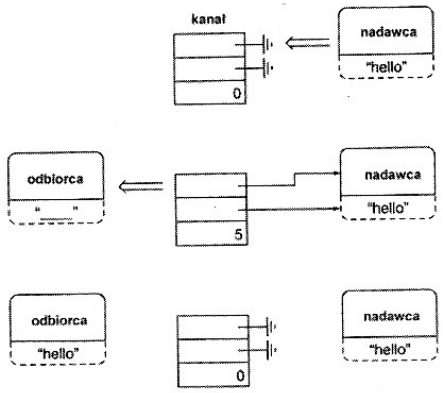
Dowolny proces może jednocześnie czytać i zapisywać do dowolnego kanału. Zawsze programista powinien zapewnić, żeby proces tylko czytał lub zapisywał do jednego kanału i aby jeden kanał był dzielony wyłącznie przez dwa procesy. Spowodowane jest to tym, że takie zjawisko zazębiania się operacji czytania i zapisywania do kanałów nie jest przewidywalne. Przykładowo, w sytuacji gdy dwa procesy próbują zapisywać, a jeden czytać, może dojść do przeplecenia się komunikatów zapisywanych do kanału w przypadkowy sposób i proces zapisujący, który próbowałby czytać z kanału mógłby się pogubić. Jeśli wiele procesów dzieli w programie kanał, lub kanał jest używany do dwukierunkowej komunikacji, taki błąd w programie może się zdarzyć, zwłaszcza, gdy program jest modyfikowany lub rozbudowywany. Takie błędy są bardzo trudne do wykrycia.

Procesy wysyłające komunikaty przez kanały używają operatora <-, po którego lewej stronie podaje się nazwę kanału, a po prawej rozdzielone przecinkami wyrażenia. Wyrażenia są obliczane i wysyłane przez kanał, przykład:

```
type (chan), auto:: c
real              :: a = 3.14
.
.
.
c <- 1, a, "tekst"
```

Tak wygląda wysłanie przez kanał c liczby całkowitej 1, prostej definicji liczby pi i krótkiego tekstu.

Proces z drugiego końca kanału może odebrać komunikat przez skopiowanie wartości na zewnątrz



Rysunek 1. Prawidłowe przesłanie komunikatu.

kanału, poprzez użycie operatora ->:

```
integer     :: i
real  :: x
type (string), auto     :: s
.
.
.
c -> i, x, s
```

Tutaj wartości z kanału są przypisywane zmiennym i, x i s. Zwróćmy uwagę na użycie *auto* w deklaracji ciągu znakowego s. Jest to spowodowane odpowiedzialnością programisty za alokacje odpowiedniego obszaru pamięci na dane odbierane z kanału. Bez tej deklaracji, lub bez ewentualnego jawnego wywołania polecenia alokującego odpowiedni obszar pamięci, kanał mógłby stwierdzić, że zmiennej s nie można skojarzyć z wartością w kanale i zgłosić błąd.

Kanały mają wiele typów danych, które mogą być nimi przesyłane, dowolny komunikat może być przesyłany przez dowolny kanał. System musi na bieżąco kontrolować, czy typy wartości wysyłanych odpowiadają typom wartości, które mają zostać odebrane.

```
type (chan), auto:: c
.
.
.
create
  1 of  sender(c)
  1 of receiver(c)
end create
.
.
.
procedure sender(c)
  type(chan)       :: c
  integer    :: a(3), i
  c <- a, i
end procedure
.
.
.
procedure receiver(c)
  type(chan)       :: c
  real       :: x, a(2)
  c-> a, i
end procedure
```

Po kompilacji i uruchomieniu powyższego kawałka kodu powinien wystąpić błąd, gdyż typy i rozmiar wartości wysyłanych i odbieranych nie są ze sobą zgodne.

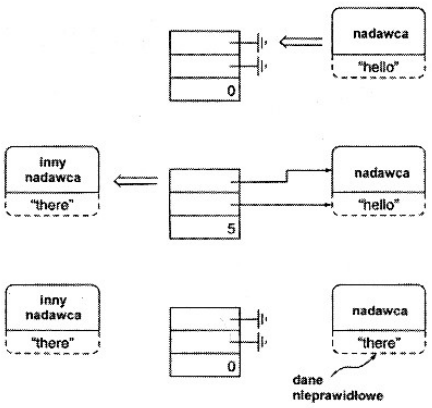
Kanały pełnią dwie ważne funkcje. Pierwsza – komunikacyjna, uruchomione procesy mogą wymieniać informacje wyłącznie przez kanały. Druga, równie ważna, dotyczy synchronizacji, komunikaty są przesyłane wyłącznie wtedy, gdy obydwa uczestniczące procesy zadeklarują swoją gotowość. Jeżeli nadawca podjął transmisję pierwszy, jest blokowany dopóki inny proces nie będzie chętny do odebrania komunikatu z kanału. I na odwrót, jeśli odbiorca jest pierwszy, jest blokowany, dopóki wiadomość nie będzie gotowa do odbioru.

Ponieważ istnieje wyżej opisane zjawisko, kanał może być używany jak binarny semafor do synchronizacji pary procesów. Znakomicie można wykorzystać jego właściwości, w ten sposób, że jeden proces wysyła komunikat i czeka na potwierdzenie, drugi proces, odbiorca, wykonuje jakąś pracę, po której wysyła komunikat potwierdzenie i synchronizuje się z procesem, który na to czekał.

Rozważmy teraz jedną z możliwych implementacji kanałów. Każdy kanał jest reprezentowany w pamięci przez rekord składający się z trzech słów. Proces, który chce przesłać dane przez kanał zaczyna od sprawdzenia pierwszego słowa w rekordzie. Jeśli to słowo wynosi *null*, oznacza to, że żaden inny proces nie komunikuje się przez kanał. Nadawca wypełnia wtedy rekord wskaźnikiem na siebie, na dane, które chce wysłać i licznikiem ilości bajtów, które chce wysłać. Proces oczekuje na odebranie danych przez odbiorcę. Trochę później, inny proces próbuje czytać dane z kanału. Najpierw sprawdza rekord statusu kanału i dostrzega, że proces nadający czeka na odbiorcę.

Odbiorca może skopiować dane wskazywane przez drugie słowo rekordu informacyjnego kanału, od długości określonej przez trzecią wartość tegoż rekordu do własnej pamięci. Następuje wznowienie pracy procesu nadawcy na skutek działania odbiorcy, który posługuje się pierwszym polem rekordu, zawierającym jego adres. Wszystkie wyżej opisane etapy obrazuje Rysunek 1.

Oczywiście, ten schemat jest symetryczny, jeśli odbiorca pojawi się pierwszy, wpisuje wskaźnik na siebie, wskaźnik na obszar pamięci gdzie mają znaleźć się odbierane dane i ilość danych, jaką jest w stanie pomieścić w buforze odbiorczym. Nadawca może stwierdzić, że jego partner, odbiorca, czeka i może przekopiować odpowiednie wartości. Problem pojawia się, gdy dwa procesy próbują wysyłać do tego samego kanału. Pierwszy wypełnia rekord statusu kanału jak wcześniej, a drugi napotyka na już wypełniony rekord i zakłada, że tym pierwszym procesem był odbiorca. W skutek tego błędnego założenia nadpisuje dane pierwszego procesu – nadawcy. W odwrotnym wypadku, gdy dwa procesy próbują odczy-



Rysunek 2. Nieprawidłowe przesłanie komunikatu.

tać z tego samego kanału, drugi może skopiować bufor, do którego pierwszy próbował odebrać dane (patrz Rysunek 2).

Są dwie drogi do rozwiązania tego problemu. Pierwsza polega na dodaniu większej ilości pól do rekordu statusu kanału, które by pozwalały stwierdzić czy będący w toku proces próbuje wysyłać lub odbierać. W ten rodzaj zabezpieczenia wyposażony jest Fortran-K (CSP), ale Occam tego nie posiada. Druga możliwość polega na rozróżnianiu końcówek kanałów do zapisu i odczytu, wtedy pomyłki można wyłapać już w procesie kompilacji. W tym schemacie kanały z końcówkami do zapisu i odczytu mogą mieć następującą postać:

```
type chan
  type(chan_reading_end) :: rd
  type(chan_writing_end) :: wr
end type
```

Operator wysyłania <- może być użyty tylko w stosunku do zapisującego końca kanału, a operator odbierania -> tylko na końcówce do odczytu.

Protokół taki, jaki został opisany wcześniej pozwala procesom wysyłać komunikaty z różnymi identyfikatorami komunikatów do jednego kanału, ale mechanizm ten jest daleki od zezwolenia procesom na wybieranie komunikatów z różnych kanałów. W tym celu wprowadzono specjalną konstrukcję dającą taką możliwość.

```
alt
  choose c1 -> ...
     {operacja 1}
  end
  choose c2 -> ...
     {operacja 2}
  end
.
.
.
  choose cn -> ...
     {operacja n}
  end
end alt
```

Po kolei następują próby odebrania komunikatów z występujących po instrukcji choose kanałów, jeśli udaje się odebrać komunikat, zostaje wykonana operacja znajdująca się w bloku choose – end i następuje przejście do końca bloku wyboru kanału. W przypadku braku możliwości odbioru sygnalizowany jest błąd, sprawdzanie jest kontynuowane dalej w sposób cykliczny. Istnieją różne modyfikacje instrukcji alt, dodające instrukcje warunkowe czy upraszczające zapis.

Istnieje konstrukcja umożliwiająca dodanie instrukcji warunkowych. Najpierw są sprawdzane warunki, a dopiero, jeśli są spełnione następuje sprawdzenie czy komunikat został nadany.

```
logical :: L = .true.
alt
  choose c1 -> ... when(L=.true.)
  {operacja 1}
  end
  choose c2 -> ... when(L=not.true.)
{operacja 2}
end
.
.
.
choose cn -> ...
{operacja 3}
end
end alt
```

możliwe jest uproszczenie zapisu do następującej postaci:

```
alt
  any j = 1, n choose cj -> ...
  end
end alt
```

co jest równoznaczne z zapisem

```
alt
  choose c1 ->
  end
  choose c2 ->
  end
.
.
.
  choose cn ->
  end
end alt
```

Istnieje również wariant dopuszczający następującą konstrukcję:

```
logical :: .true.
alt
  choose c -> ... when L =.true.
  {operacja}
    L=.false.
  when L=.false.
    {inna operacja}
  end
end alt
```