CD-ROM: KEA! 420 5.1 i KEA! 340 5.1, XFree86 3.3.5, GIMP 1.0.4 i inne

12/99

magazyn

Linux & UNIX

Polonizacja LaTeX-a

NDS dla Solarisa

Konfiguracja serwera Apache

TCP/IP od podstaw

Linda kontra PVM

Tworzenie Wirtualnych Sieci Prywatnych

Programowanie w języku PostScript

CD ROM

NR 12(16)/99 GRUDZIEŃ CENA 17 ZŁ ISSN 1506-1299

magazyn Linux & UNIX

Adres kontaktowy: magazyn@tao.com.pl

E-commerce w świecie Unixa na przykładzie InterMarket 3D (część IV)..... 5

Polonizacja LaTeX-a..... 8
Adam Dawidziuk
Piotr Bolek

XV Międzynarodowe Targi KOMPUTER EXPO - 2000..... 11

Programowanie w języku PostScript - część III..... 12
Adam Dawidziuk

Produkty

KEA! 420 i KEA! 340 wersja 5.1..... 16

Programowanie

JavaScript w praktyce - część I..... 19
Wiktor Łukasik

Inteligentne aplikacje wielkiej stopy..... 25
Marek Sawerwain

Integracja

ViewNow Suite. Kompletna integracja stacji Windows z systemami odległymi w sieciach SNA i IP..... 28

Technologie Sieciowe

Administrowanie mieszanymi sieciami NetWare i Solaris za pomocą systemu NDS 31
Vinita Kurup

TCP/IP od podstaw - część III..... 36
Wojciech Bojdoł

Linda kontra PVM?
Linda - implementacja modelu z komunikacją generowaną..... 38
Piotr Koch

Tworzenie Wirtualnych Sieci Prywatnych w oparciu..... 42
o oprogramowanie CIPE
Tomasz Zawalski

Linux dla każdego

Podstawowa konfiguracja serwera HTTP Apache..... 45
Wojciech Bojdoł

Podłączenie do Interentu za pomocą terminala ISDN..... 46
Wojciech Bojdoł

Filozofia RPMa - część II..... 48
Michał Kuratczyk

Tips & Tricks

Niskopoziomowe formatowanie dyskietek w Linuksie..... 54
Janusz Wirtz

Linda kontra PVM ?

Linda – implementacja modelu z komunikacją generowaną

Piotr Koch

Jest to system bazujący na koncepcji wspólnej przestrzeni roboczej nazywanej przestrzenią krotek (ang. *tuple space*), służącej do komunikowania się uruchomionych procesów w ramach tego systemu. Przestrzeń krotek jest związana z modelem przetwarzania rozproszonego zwanego komunikacją generowaną (ang. *Generate Comunication*) alternatywą do modelu z przesyłaniem komunikatów (ang. *Message Passing*) i pamięcią dzieloną (ang. *Shared Memory*).

Dostęp do rozproszonej pamięci dzielonej odbywa się za pomocą małego zbioru elementarnych operacji, które można dodać do istniejących języków, takich jak C lub Fortran, tworząc w ten sposób języki równoległe: C-Linda i Fortran-Linda. Podejście takie ma wiele zalet w porównaniu z zaprojektowaniem nowego języka programowania. Podstawową zaletą jest to, że użytkownicy nie muszą się uczyć nowego języka. Zalety tej nie należy niedoceniać. Drugą zaletą jest prostota: język X można przekształcić w język X-Linda przez dołączenie do biblioteki kilku nowych operacji elementarnych i zaadaptowanie preprocesora systemu Linda, który przekazuje programy systemu Linda do kompilatora. System Linda jest także łatwo przenośny między systemami operacyjnymi i architekturami maszyn, i został zrealizowany w wielu systemach rozproszonych i równoległych. Podstawowe cechy pakietu Linda:

- Równoległy język programowania bazujący na C (C-Linda) i Fortranie (Fortran-Linda)
- Możliwość łączenia Lindy z programami w C i Fortranie
- Tworzenie programów równoległych na platformy przetwarzające odległe od siebie
- Łatwa w użyciu

- Bazuje na logicznie globalnej asocjacyjnej pamięci
- Implementacja równoległości przy użyciu małej liczby operacji operujących na przestrzeni krotek w celu tworzenia i koordynowania procesów

Unowocześnienie systemu *Linda*, jest znane pod nazwą *Pirhana* lub też *Linda - Pirhana System*. Zasadniczą modyfikacją jest to, że wprowadzono lokalizację wykonywanych zadań we wspólnym obszarze roboczym oraz możliwość monitorowania wykonywanych procesów za pomocą programów narzędziowych reprezentujących informacje w przyjaznej formie graficznej (ang. *active agents*). Aktualnie, nowe wersje tego systemu mogą być również implementowane na wielu różnych platformach co znacznie podnosi jego atrakcyjność użytkową.

Podstawowe operacje na przestrzeni krotek

Ujednolicającą koncepcją systemu Linda jest abstrakcja zwana przestrzenią krotek (n-tek, ang. *tuple space*). Przestrzeń krotek jest globalna w całym systemie. Procesy mogą wstawiać i usuwać krotki z przestrzeni krotek niezależnie od tego, gdzie się je przechowuje. Dla użytkownika przestrzeń krotek przedstawia się jako duża, globalna pamięć dzielona. Rzeczywista implementacja może obejmować wiele serwerów na wielu maszynach.

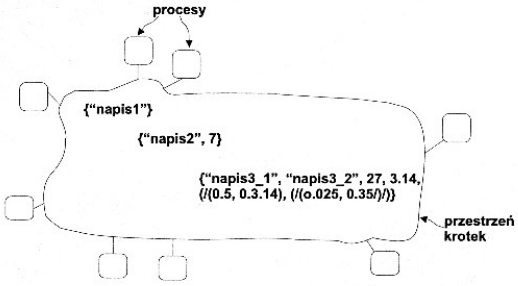
Krotka (ang. *tuple*) przypomina strukturę języka C lub rekord języka Pascal. Składa się z jednego lub więcej pól o typach dopuszczalnych w bazowym języku. W języku C-Linda pola mogą mieć typy liczb całkowitych, długich liczb całkowitych, liczb zmiennopozycyjnych oraz typy złożone, takie jak tablice (włącznie z napisami) i struktury (lecz bez innych krotek). Każde pole może zawierać konkretne wartości, mówimy wtedy, że jest rzeczywiste (ang. *actual*), lub może być nieokreślone, mówimy wtedy o polu formalnym. Pola formalne są używane do tworzenia wzorów dobierania.

Przykładowe krotki zawierające wyłącznie pola rzeczywiste mogą wyglądać następująco:

```
{"napis1"}
{"napis2", 7)
{"napis3_1", "napis3_2", 27, 3.14, {/{/
0.5,0.3.14/),{/{/0.025,0.35/)/)}
```

Pierwsza z krotek zawiera ciąg znaków "napis1", druga ciąg "napis2" i liczbę całkowitą 7, a trzecia zawiera dwa ciągi, liczbę całkowitą 27, rzeczywistą 3.14 i tablicę liczb rzeczywistych o rozmiarze 2 na 2 pola. Krotki istnieją jedynie w przestrzeni krotek, która nie przechowuje nic oprócz nich, gdyż same procesy znajdują się na zewnątrz, co pokazuje Rysunek 1.

Procesy dodają krotki do przestrzeni krotek używając operacji *out*. Operacja ta pobiera jako argumenty jedno lub więcej wyrażeń, *out* oblicza te wyrażenia w nieokreślonym porządku i tworzy krotkę zawierającą wyniki tych obliczeń. Przykładowy kod w języku For-



Rysunek 1: Przestrzeń krotek.

tran wykorzystujący wcześniej przytoczone przykłady krotek może wyglądać następująco:

```
type(string)            :: label_1="napis1"
integer, constant       :: i=7
type(string), auto      :: j
real                    :: block(2, 2)
.
.
.
label_2="napis2"
j=27
block(1, 1)=0.5; block(1, 2)=3.14
block(2, 1)=0.025; block(2, 2)=0.35
.
.
.
out label_1
out label_2, i
out "napis3_1", "napis3_2", j, 3.14, block
```

Operacje zapisywania do pamięci nie wystarczą do stworzenia programu, dlatego model z komunikacją generowaną przewiduje operacje do czytania i usuwania krotek. Pierwsza z nich nazywa się *rd*. Tak jak *out*, *rd* bierze jedno lub więcej wyrażeń jako argumenty, ale zamiast dodawać krotkę do przestrzeni krotek, *rd* blokuje proces wywołujący dopóki nie zostanie dopasowana krotka w przestrzeni krotek zgodnie argumentami operacji. Wywołanie operacji *rd* może wyglądać następująco:

```
rd "napis1"
```

kiedy przestrzeń krotek zawiera tylko trzy wyżej przedstawione w przykładach krotki, możliwe jest wywołanie:

```
rd "napis4"
```

Przestrzeń krotek nie zwiera krotki "napis4", więc krotka nie mogła być dopasowana. Proces jest zablokowywany dopóki inny proces nie umieści odpowiedniej krotki w przestrzeni.

Drugą operacją czytającą krotki jest operacja *in*, która działa jak *rd*, ale, gdy krotka zostanie dopasowana następuje usunięcie jej z przestrzeni krotek. Operacja *in* na konkretnej krotce może być wykonywana tylko raz, w odróżnieniu od operacji *rd*, która na jednej krotce może być wykonywana wielokrotnie. Procesy mogą się synchronizować używając krotek w podobny sposób jak używa się semaforów.

Problem z instrukcjami *in* i *rd* polega na tym, że proces musi dokładnie wiedzieć co chce pobrać z przestrzeni krotek zanim instrukcja pobierania krotki zostanie wykonana. Jest to niewygodne założenie, gdyż program musi znać poszukiwaną wartość zanim odczyta ją z przestrzeni krotek, dlatego w komunikacji generowanej istnieje pojęcie antykrotki (ang. *anti-tuples*). Antykrotka jest specyficznym rodzajem krotki, w której część pól jest polami formalnymi, to znaczy mającymi określony typ, ale nie posiadającymi wartości. Pola formalne w antykrotce oznacza się znakiem zapytania postawionym przed zmienną lub typem, na przykład:

```
integer             :: i, b(4)
type(string)        :: s
.
.
.
in "napis1", ?i     ! drugie pole jest formalne
rd ?s, 3, ?b        ! pierwsze i trzecie pole są
                    ! formalne
```

Jeśli proces używa antykrotki do operacji czytania, użyta zostaje dowolna krotka zgodna z antykrotką. Krotka i antykrotka są zgodne jedynie gdy:

- Mają tą samą ilość pól
- Typy odpowiednich pól są takie same
- Wartości odpowiednich pól rzeczywistych są takie same

Tabela 1. Przykłady zgodności krotek.

integer :: i=3, j=4, X(3), Y(2, 2), Z(4)
logical :: b

Antykrotka	Krotka	Zgodność	Komentarz
{2}	{2}	tak	wartości zgodne
{2}	{3}	nie	wartości niezgodne
{i}	{3}	tak	typy i wartości zgodne
{i}	{2}	nie	typy zgodne, ale wartości niezgodne
{?i}	{2}	tak	typy zgodne (i przyjmuje wartość 2)
{?i, ?b}	{2, .true.}	tak	typy zgodne
{?b, ?i}	{2, .true.}	nie	typy w niezgodnym porządku
{X}	{(/1, 2, 3/)}	błąd	niewłaściwe porównanie
{?X}	{(/1, 2, 3/)}	tak	typ i rozmiar zgodne
{?X}	{(/1, 2, 3, 4/)}	nie	nieudane porównanie rozmiaru
{?Y}	{Z}	nie	nieudane porównanie wymiaru

Zasady porównywania krotek wyjaśnia na przykładach Tabela 1.

Jeśli operacja *rd* lub *in* nie może znaleźć pasującej krotki do antykrotki, proces wykonujący operację jest blokowany dopóki operacja dopasowania nie jest możliwa.

Krotki nie są przechowywane tak jak inne struktury danych, wewnątrz procesów. Są one natomiast przechowywane w odseparowanej przestrzeni krotek, która ma cztery ważne własności: jest dzielona, wszystkie procesy pracują z tą samą przestrzenią krotek i każdy proces ma do niej dostęp; nie jest zbiorem, ale "workiem" (ang. *bag*), w którym może się znajdować wiele identycznych krotek; jest asocjacyjna, krotki są usuwane z przestrzeni krotek przy pomocy wzorców porównawczych, według zasad dopasowywania przedstawionych wcześniej; jest anonimowa, gdy raz umieścimy krotkę, system nie potrafi stwierdzić kto i kiedy ją stworzył. Program może umieszczać informacje o pochodzeniu krotki w niej samej, ale za zarządzanie tymi informacjami jest odpowiedzialny programista.

Ostatnią z podstawowych operacji w modelu z komunikacją generowaną jest operacja *eval*. Ma ona ten sam końcowy rezultat co *out*, umieszcza krotkę w przestrzeni krotek, ale osiąga to w okrężną drogą. Zamiast obliczenia wartości każdego pola i stworzenia krotki w celu przechowania wyników, *eval* tworzy aktywną krotkę (ang. *active tuple*), która tworzy nowy proces w celu obliczenia każdego pola krotki. Każdy z tych procesów działa tak długo dopóki nie obliczy wszystkich pól w aktywnej krotce. Gdy zostaną wypełnione wszystkie pola, krotka jest przekształcana w zwykła krotkę, zwaną też pasywną, taką jaką tworzy się operacją *out*. Przykładowe utworzenie krotki aktywnej może wyglądać następująco:

```
eval "przykład", x, f(x), g(x, y, z)
```

gdzie f(x) i g(x, y, z) są wcześnie zdefiniowanymi funkcjami. Stworzony w ten sposób proces jest równy swojemu stwórcy, nie ma żadnych połączeń między nim a jego rodzicem i może wykonywać operacje na krotkach włącznie z operacją *eval*.

Model zwielokrotnionego pracownika

Powszechnie stosowanym w systemie Linda paradygmatem jest model zwielokrotnionego pracownika (ang. *replicated worker model*). Oparty on jest na koncepcji worka zadań (ang. *task bag*) pełnego prac do wykonania. Proces główny rozpoczyna działanie od wykonania pętli zawierającej operację

```
out "worek-zadań", zadanie
```

w której przy każdym powtórzeniu wprowadza się do przestrzeni knotek opis innego zadania. Każdy pracownik rozpoczyna od pobrania za pomocą operacji `rd "worek-zadań", ?zadanie` krotki z opisem zadania, które zaczyna wówczas wykonywać. Po zakończeniu zadania pracownik pobiera następne. Nowe prace mogą nadchodzić do worka również podczas wykonywania. W ten prosty sposób niewielkim nakładem pracy następuje dynamiczny podział zadań między pracowników i każdy z nich jest nieustannie zajęty.

Tabela 2. Maszyny obsługiwane przez system Network Linda.

Komputer	Opis
Sparc	SunOS 4.1 i SunOs 5.2
Dec Alpha	OSF 1.3
DECStation	
RS/6000	
SGI	
HP 9000/700	

Ogólne zasady programowania w systemie Linda opierają się na modelu *Master/Worker* z pamięcią dzieloną (Rysunek 2) i polegają na tym, że:

- Zadania i workery są niezależne względem siebie
- Master dzieli prace na dyskretne zadania i umieszcza w globalnej przestrzeni
- Workery bezustannie pobierają zadania i zwracają wyniki z powrotem do globalnej przestrzeni
- Master pobiera wyniki z globalnej przestrzeni (Rysunek 2)

Network Linda

Network Linda jest implementacją równoległego modelu programowania – Lindy stworzoną przez SCA (Scientific Computing Associates), która może być uruchomiona na klastrach stacji roboczych o niekoniecznie takich samych architekturach. Dostarczane są dwie wersje kompilatorów, C-Linda i FORTRAN-Linda. Dodatkową własnością Networ Linda są dwie rozszerzające standard funkcje, *pin* i *prd*, które są nie blokującymi modyfikacjami operacji *in* i *rd*.

Network Linda potrafi obsługiwać sieci heterogeniczne. Obsługiwane typy maszyn przedstawia Tabela 2. Do przenoszenia danych miedzy maszynami o różnych ich reprezentacjach używana jest opcja konwersji danych do formatu Sun XDR.

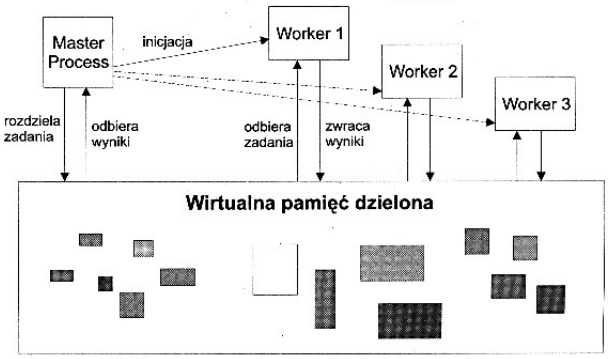
Network Linda jest systemem komercyjnym i w pełni płatnym. Wraz z systemem dostarczane są dwa kompilatory C-Linda i FORTRAN-Linda. CDS (ang. *Code Developent System)* pozwala tworzyć kod programu, wykonywać i symulować równoległość na pojedynczych stacjach roboczych, dostarcza ponadto bardzo użyteczne narzędzia do debugowania.

Tworzenie i kontrola procesów

W systemach z komunikacją generowaną tworzenie procesów odbywa się w wyniku wykonania operacji *eval*, która tworzy krotkę aktywną. Pola w krotce, która jest tworzona przez instrukcje *eval*, są obliczane przez konkurencyjne, odseparowane od siebie procesy. Jeśli wszystkie pola zostaną obliczone, pasywna krotka zostaje umieszczona w przestrzeni krotek. Typowe użycie tej instrukcji może wyglądać w następujący sposób:

```
eval("Tekst", x, f(x));
```

Wykonanie takiej operacji powoduje stworzenie krotki aktywnej, której wartość pola f(x) jest obliczana, przez specjalnie powołany do tego, przez system Network Linda, proces. Programista nie ma wpływu na to, gdzie proces ten zostanie uruchomiony, tak jak na przykład w PVM. System Linda sam odnajduje najbardziej odpowiedni według niego procesor i tam powołuje go do życia. Do zadań programisty nie należy troska gdzie proces ma być uruchomiony, czy kod zdalnego procesu jest dostępny na mającym wykonać go komputerze, ani czy operacja zdalnego uruchomienia procesu powiodła się. W najprostszym przypadku, do zmartwień programisty nie należy nawet kończenie tego procesu. Proces jest automatycznie kończony w momencie gdy obliczy przydzielone mu pole krotki. Gdy wszystkie pola krotki zostają obliczone, zostaje ona przekształcona w krotkę pasywną i umieszczona w przestrzeni krotek, skąd program główny może ją pobrać, w celu poznania wyników obliczeń.



Rysunek 2: Model programowania Master/Worker z użyciem wirtualnej pamięci dzielonej.

Jak widzimy Network Linda oferuje dużo większą prostotę niż inne biblioteki do przetwarzania rozproszonego, takie jak PVM, MPI czy P4. Programy tworzone w Lindzie są pisane niemal jak programy sekwencyjne. Programista nie musi określać, jak w przypadku PVM, adresów hostów czy architektur na jakich mają być powoływane procesy. Nie musi też dbać o przestrzeganie określonych zasad przy tworzeniu części kodu, która ma być wykonywana równolegle. Tworzy po prostu kod funkcji jak w przypadku sekwencyjnego programu i poprzez wykonanie funkcji *eval* zleca systemowi Linda, że obliczenia mają być wykonane równolegle. Do samego systemu należy odpowiednie powołanie procesu, jego inicjacja i zakończenie jego działalności po dotarciu do końca obliczeń. Prostota ta powoduje, że tworzony kod programu jest prostszy, bardziej czytelny i trudniej o pomyłkę. PVM umożliwia pewnego rodzaju automatyzację wybierania maszyn, na których są wykonywane zdalne procesy, ale za całą resztę związaną z prawidłowym działaniem procesów zdalnych i powoływaniem ich przez program główny, i prawidłowym kończeniem ich działania odpowiedzialna jest osoba kodująca. Ta cecha PVM'a i wielu innych systemów tego typu nie jest jednak pozbawiona zalet. Owszem wymaga dużo większego nakładu pracy i uwagi podczas programu, ale przy dysponowaniu dużą różnorodnością maszyn tworzących wirtualny komputer pozwala na zoptymalizowanie programu przez uruchamianie procesów na bardziej odpowiednich komputerach i architekturach.

W Lindzie tworzone procesy są anonimowe, nie posiadają identyfikatorów. Identyfikacja tych procesów, w zasadzie nie jest potrzebna, tak jest pomyślane przetwarzanie w tym modelu. Jeśli jednak jest to potrzebne, procesy mogą być znaczone przez programistę poprzez dodanie dodatkowych pól w tworzonych krotkach aktywnych, które będą zawierać przydzielone im przez programistę identyfikatory. Samodzielne nazywanie procesów może oczywiście prowadzić do błędów w wyniku na przykład nadania dwóm procesom tych samych identyfikatorów. Jednak praktykuje się coś takiego powszechnie, zwłaszcza w programach wykorzystujących wcześniej opisywaną technikę *Master/Worker,* gdzie procesy tworzone są jednokrotnie, zostają im przydzielone etykiety, które determinują później, które krotki z danymi/zadaniami są dla nich przeznaczone. Programista musi w takim wypadku sam zadbać, aby po zakończeniu obliczeń wysłać odpowiednie krotki do wszystkich procesów, oznajmiające im, że mają zakończyć swoje działanie (sam kod funkcji, będącej parametrem instrukcji *eval* powołującej workery powinien być tak napisany, aby cały czas pobierał krotki z danymi obliczał je, a jednocześnie oczekiwał takiej, która oznacza zakończenie obliczeń).

Ograniczeniami funkcji obliczanych w wyniku wykonania operacji *eval* jest to, że mogą mieć do szesnastu argumentów, a zwracane wartości muszą należeć do poniższego zbioru typów:

- C-Linda: int, long, short, char, float, double
- Fortran-Linda: integer,real, double precision, logical

Poniższe przykłady pokazują wykonanie pętli, umieszczającej krotki w przestrzeni krotek, w których istnieje potrzeba obliczenia wartości części ich pól. W pierwszym wypadku obliczenia odbywają się w sposób sekwencyjny, wartości pól są obliczane i dopiero wtedy krotka pasywna jest umieszczana w przestrzeni krotek (użycie instrukcji *out*):

```
for (i=0; i<100; i++)
  out("Wartość funkcji f(i)", i, f(i));
```

W drugim przypadku w wyniku operacji *eval,* tworzone są krotki aktywne, których pola są obliczane w sposób równoległy, przez konkurencyjnie działające, utworzone tą funkcją procesy:

```
for (i=0; i<100; i++)
  eval("Wartość funkcji f(i)", i, f(i));
```