CD-ROM: Polska edycja TeX Live 5!

5/2000

magazyn

Linux & UNIX

HIS a Linux

Debian GNU/Linux

LilyPond

Nowe trendy i technologie w GNOME

TCP/IP w praktyce

Caldera OpenLinux eServer 2.3

Nowe produkty Allied Telesyn

CD ROM

NR 5(21)/2000 MAJ CENA 17 Zł ISSN 1506-1299

Adres kontaktowy: magazyn@tao.com.pl

Organizacja komunikacji w systemach obiektowych opartych na OMG CORBA 8
Jędrzej Byrski

Serwis tolerowania uszkodzeń dla aplikacji opartych na architekturze CORBA 12
Jędrzej Byrski

Debian/GNU z perspektywy administratora - część I 17
Grzegorz Nalepa

LilyPond 24
Krzysztof Leszczyński

Produkty

Caldera OpenLinux eServer 2.3 30
Wojciech Bojdoł

Nowe urządzenia sieciowe Allied Telesyn 32
Opracował Marcin Koprowski

Check Point 2000 34

Programowanie

Nowe trendy i technologie w GNOME 38
Marek Sawerwain

Technologie sieciowe

HIS a Linux 41
Bartosz Maruszewski

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

System P4 42
Piotr Koch

Linux dla każdego

C-Media CMI8330 a Linux 47
Maciej Broniarz

Tworzenie List dystrybucyjnych przy użyciu oprogramowania Listar 49
Wojciech Bojdoł

Porównanie przeglądarek tekstowych - uzupełnienie 50
Wojciech Bojdoł

Ustalanie masek sieciowych 50
Wojciech Bojdoł

Wdrożenia

System Informatyczny Polskiej Agencji Prasowej 52

Tips & Tricks

Tworzenie bootowalnych płyt CD 53
Wojciech Bojdoł

OracleMobile.com - portal internetowy dla użytkowników urządzeń internetowych 53

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

magazyn Linux & UNIX

System P4

Piotr Koch

P4 jest biblioteką makr i procedur stworzonych przez Argonne National Laboratory w celu wspierania programowania maszyn równoległych różnego typu. P4 opiera się na dwóch technikach programowania, monitorów dla systemów z pamięcią dzieloną i przesyłania komunikatów dla systemów z pamięcią rozproszoną, możliwe jest również łączenie tych dwóch technik. P4 pomyślany jest w ten sposób, aby był przenośny, wydajny oraz łatwy do instalacji i w użyciu. System ten może być użyty do programowania sieci stacji roboczych, zaawansowanych superkomputerów równoległych takich jak, na przykład Intel Touchstone Delta czy Alliant Campus (system oparty na HiPPI) oraz pojedynczych multiprocesorów z pamięcią dzieloną. Jak możemy zobaczyć w Tabeli 2 lista obsługiwanych przez P4 maszyn jest długa (*Tabela 1*). Podstawowe cechy systemu P4 to:

- Wsparcie dla przesyłania komunikatów i operacji na pamięci dzielonej
- Kodowanie XDR dla sieci heterogenicznych
- Dokumentacja w formacie programu Emacs
- Obsługa SYSV IPC dla wielozadaniowości w systemach z pamięcią dzieloną w przypadku stacji roboczych wyposażonych w wiele procesorów i możliwość symulacji na komputerach jednoprocesorowych
- Automatyczne lub kontrolowane przez użytkownika zarządzanie buforami przy przesyłaniu komunikatów
- Opcjonalny serwer P4 do szybkiego uruchamiania aplikacji na zdalnych maszynach

Obsługa współdzielonej pamięci w systemie jest łatwa, co jest charakterystyczne dla tego pakietu. Interesującą cechą jest również to, że potrafi on badać topologię sieci na której została założona maszyna wirtualna i optymalizować działania pod tym kątem. Rozszerzoną wersją systemu *P4*, jest system *Parmacs*.

Uruchamianie procesów zdalnych

Zanim zostanie wywołana którakolwiek funkcja systemu P4, aplikacja musi zainicjować środowisko wywołując funkcje:

```
int p4_initenv( int argc, char
*argv)
```

Charakter tej funkcji, może się wydawać podobny do *pvm_mytid()* systemu PVM, ale takie stwierdzenie nie jest do końca prawdą. Podstawowym zadaniem tej funkcji jest zainicjalizowanie środowiska, co pozwala między innymi dowiedzieć się systemowi P4 z jakimi argumentami został wywołany program. Wywołanie tej funkcji jest konieczne i jednokrotne, musi być wywołana przez program główny i nie może być pominięta tak jak można było to zrobić w przypadku *pvm_mytid()*. Dopiero po wywołaniu tej funkcji mogą być utworzone procesy zdalne. Istnieją do tego dwie funkcje:

```
int p4_create( int (*fxn)( void ) )
int p4_create_procgroup( void )
```

Użycie *p4_create()* oznacza, że użytkownik sam będzie zarządzał utworzonymi w ten sposób procesami. Parametrem jest wskaźnik na funkcję na podstawie, której zostanie utworzony pojedynczy proces. Nowy proces, może dzielić obszar danych (w pamięci dzielonej) z procesem-rodzicem (ang. *parent process*). Utworzony w ten sposób proces nigdy nie jest zarządzany przez P4, nie jest mu przypisywany iden-

Tabela 1. Komputery obsługiwane przez system P4.

Komputer
Sun-3, Sun386i, Sparc-1 lub stacje robocze Sparc-2
Stacje robocze Sun uruchomione na Solaris
HP workstations
Dec 5000 workstations
Komputery bazujące na 68030 lub 68040 NeXT workstations
IBM RS 6000 series workstations
IBM 3090 uruchomiony z wersją IBM UNIX, AIX
Sequent Symmetry shared-memory multiprocessor
Sequent Symmetry shared-memory multiprocessor PTX OS
Encore Multimax shared-memory multiprocessor
BBN GP-1000
BBN TC-2000
BBN TC-2000 z biblioteką do przesyłania komunikatów TCMP
Intel IPSC/860
Intel DELTA
Intel Paragon
Stardent Titan
Silicon Graphics workstations
Cray X/MP
Alliant FX/8
Alliant FX/2800 lub FX/800
Alliant FX/2800 lub FX/800, z przełącznikiem CAMPUS HiPPI
Kendall Square KSR-1
Thinking Machines' CM-5
IBM SP-1

Tabela 2. Typy danych przesyłanych przez system P4.

Parametr *datatype*	Wartość kodowana
P4INT	Liczba całkowita
P4DBL	Liczba zmienno przecinkowa podwójne precyzji
P4FLT	Liczba zmienno przecinkowa
P4LNG	Liczb całkowita długa
P4NOX	Bez translacji

tyfikator, nie może wysyłać komunikatów itd. Jedyną procedurą P4, która współpracuje z takimi procesami jest właśnie *p4_create()*.

Konstrukcja *p4_create_procgroup()* pozwala na zarządzanie procesem przez P4, oznacza to, że procesom zostają automatycznie przepisane unikatowe identyfikatory (rozpoczynające się od zera, przy czym zero przysługuje procesowi głównemu ang. *big master*, jak widzimy mamy tu do czynienia zupełnie z innym systemem nadawania identyfikatorów niż w systemie PVM), alokowane są kolejki komunikatów, aby możliwe było przesyłanie komunikatów, oraz możliwe się staje uruchamianie tych procesów na odseparowanych multiprocesorach z pamięcią dzieloną. Jedyną niedogodnością funkcji *p4_create_procgroup()* jest to, że musimy stworzyć sami plik konfiguracyjny *procgroup* opisujący zbiór potrzebnych procesów zdalnych (ang. *slave processes*), za nim uruchomimy program główny (ang. *master*). Plik taki opisuje ile procesów ma zostać uruchomionych na jakich maszynach i gdzie znajduje się wykonywalny kod, który ma zostać uruchomiony na zdalnych maszynach. Przykładowy plik konfiguracyjny *procgroup* może wyglądać następująco:

```
# startuje po jednym procesie zdalnym na
maszynach sun2 i sun3
local 0
sun2 1 /home/mylogin/p4pgms/sr_test
sun3 1 /home/mylogin/p4pgms/sr_test
```

Zapis taki oznacza, że na lokalnej maszynie zostanie uruchomiony tylko program główny, a nie zostanie tam uruchomiony żaden proces roboczy, na maszynie sun2 zostanie uruchomiony jeden proces, którego kod znajduje się w następującej lokalizacji /home/mylogin/p4pgms/sr_test, podobnie na maszynie sun3. Generalnie plik konfiguracyjny ma postać:

```
lokalna_maszyna n [pełna_ścieżka_z_nazwą]
[loginname]
zdalna_maszyna n pełna_ścieżka_z_nazwa [login-
name]
...
gdzie n jest liczbą procesów, które mają być
uruchomione
```

p4_create_procgroup() czyta plik *procgroup* aby stwierdzić ile procesów ma być utworzonych i gdzie mają być umieszczone. Procedura ta najpierw sprawdza nazwę pliku wyspecyfikowanego z linii komend za argumentem *–p4pg* jeśli taki istnieje. Jeśli takiego nie ma, procedura szuka pliku o takiej samej nazwie co plik wykonywalny, tyle że z rozszerzeniem „.pg". Jeśli nie znajdzie, szuka pliku nazwanego *„procgroup"*. Buduje ona na podstawie tego pliku tablicę opisującą wszystkie utworzone procesy i przekazuje kopie do każdego procesu. Proces może użyć tej tablicy do komunikowania się z innymi (procesy w klastrze mogą wysyłać komunikaty bezpośrednio przez pamięć dzieloną poprzez mechanizmy charakterystyczne dla danego sprzętu).

Alternatywną metodą jest samodzielne zbudowanie takiej tablicy w pamięci (przez użycie funkcji *p4_alloc_procgrpup()*) i użycie *p4_setup(struct p4_procgroup *pg)* w celu ustawienia jej jako obowiązującej i powołania procesów zdalnych.

W celu zakończenia swojej działalności wszystkie procesy powinny wykonać następującą procedurę:

```
void p4_wait_for_end()
```

Jest to przerywająco-czyszcząca procedura wywoływana na końcu wykonywania każdego programu wykorzystującego system P4. W przypadku procesu głównego powoduje przerwanie jego pracy i przejście w stan oczekiwania na zakończenie pracy procesów zdalnych.

```
void p4_get_my_id()
```

Identyfikator przypisany procesowi zwraca liczbę całkowitą przez system P4. Jeśli proces nie jest zarządzany przez P4, zwracana wartość wynosi –1.

```
int p4_num_total_ids()
```

Informuje o liczbie identyfikatorów związanych z uruchomionymi przez system procesami na wszystkich klastrach, włącznie z programem głównym (ang. *big master*) i lokalnymi programami głównymi na poszczególnych klastrach (ang. *remote master*).

```
int p4_num_total_slaves()
```

Informuje o liczbie identyfikatorów związanych z uruchomionymi przez system procesami na wszystkich klastrach, włącznie z lokalnymi programami głównymi na poszczególnych klastrach (ang.*remote master*), ale bez programu głównego (ang. *big master*) .

Próbując porównać tę kategorię funkcji z odpowiednikami w PVM, widzimy, że narzucone tutaj rozwiązania uniemożliwiają dynamiczne tworzenie i usuwanie procesów zarządzanych systemem P4, tak jak miało to miejsce w przypadku PVM. Nie mamy także możliwości automatycznego zarządzania przez system rozmieszczeniem procesów zdalnych, wszystko jest skonfigurowane „na sztywno", cała odpowiedzialność za rozmieszczenie procesów spada na użytkownika. Daje to duże możliwości manipulacji, których nie jest też

pozbawiony PVM, ale zarazem poważnie ogranicza, gdyż nie pozwala podejmować żadnych działań podczas pracy systemu związanych ze zmianą obciążenia wykorzystywanych maszyn. Tak więc zarządzanie procesami w tym systemie wydaje się dużo mniej efektywne niż w opisywanych we wcześniejszych artykułach PVM i zdecydowanie bardziej skomplikowane niż miało to miejsce w systemie Network Linda. Widoczny jest nacisk na możliwość łączenia technik związanych z komunikacją poprzez dzieloną pamięć z bazującymi na przesyłaniu komunikatów.

Pobieranie informacji o stanie systemu

Do funkcji informacyjnych systemu P4 możemy zaliczyć:

```
char *p4_version()
```

Funkcja ta informuje z jaką wersją systemu P4 mamy aktualnie do czynienia.

```
void p4_print_avail_buffs()
```

System P4 utrzymuje tablice z listą buforów różnej wielkości, która może się szybko zmieniać w wyniku alokacji i zwalniania buforów. W każdej chwili możemy zobaczyć zawartość całej puli buforów wywołując tę procedurę.

```
void p4_set_avail_buff( bufidx, size)
```

Funkcja ta służy do ustawiania rozmiarów buforów stanowiących zasoby systemu P4. Parametr *bufidx*, wyszczególnia bufor w liście buforów i powinien być numerem od 0 do 7. *size* określa do jakiego rozmiaru ma być zwiększony bufor znajdujący się w liście buforów zarządzanych przez P4. Jest to potrzebne do utrzymania buforów w rosnącym porządku. Domyślnie lista rozmiarów buforów wygląda następująco: 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576.

Konfiguracja węzłów przetwarzających

Jak już zostało powiedziane wcześniej w P4 nie ma możliwości dynamicznego tworzenia ani procesów, ani grup procesów. Wszystkie informacje o tym jakie procesy tworzą grupę i gdzie się znajdują oraz na jakich maszynach mają być uruchamiane znajdują się w odpowiednich plikach *procgrup*. Plikie te opisują ile procesów ma zostać uruchomionych na jakich maszynach i gdzie znajduje się ich wykonywalny kod, który ma zostać uruchomiony na zdalnych maszynach. Przykładowy plik konfiguracyjny *procgroup* może wyglądać następująco:

```
# startuje po jednym procesie zdalnym na
maszynach sun2 i sun3
local 0
sun2 1 /home/mylogin/p4pgms/sr_test
sun3 1 /home/mylogin/p4pgms/sr_test
```

Zapis taki oznacza, że na lokalnej maszynie zostanie uruchomiony tylko program główny, a nie zostanie tam uruchomiony żaden proces roboczy, na maszynie sun2 zostanie uruchomiony jeden proces, którego kod znajduje się w następującej lokalizacji /home/mylogin/p4pgms/sr_test, podobnie na maszynie sun3. Generalnie plik konfiguracyjny ma postać:

```
lokalna_maszyna n [pełna_ścieżka_z_nazwą]
[loginname]
zdalna_maszyna n pełna_ścieżka_z_nazwą [login-
name]
...
```

gdzie n jest liczbą procesów typu „slave", które mają być uruchomione .

Plik *procgoup* może być podany jako parametr dla aplikacji głównej, która wykonuje operacje *p4_create_procgoup()*, w następujący sposób:

```
nazwa_aplikacji -p4pg nazwa_pliku_konfigura-
cyjnego
```

Gdy program nie stwierdzi takiego parametru, poszukiwany jest w macierzystym katalogu programu głównego dowolny plik z rozszerzeniem *pg*. Jeśli i taki nie zostanie znaleziony system próbuje wczytać plik „*procgroup*", jeśli i to nie zakończy się powodzeniem procesy nie zostają utworzone.

Komunikacja

Wachlarz procedur do przesyłania komunikatów w porównaniu z PVM czy standardem MPI wygląda raczej skromnie. Mniejszy wybór funkcji, nie musi oznaczać jednak gorszej użyteczności systemu, czy jego słabszej wydajności, co zdawał się potwierdzać przedstawiany już na łamach czasopisma system Network Linda. P4 wydaje się czymś pośrednim między dużymi możliwościami biblioteki PVM, a prostotą systemu Linda. Zbiór istniejących w P4 procedur wysyłających komunikaty, pozwala na wysyłanie komunikatów w sposób blokujący lub nie blokujący, z kodowaniem XDR lub bez. Przesyłanie może odbywać się z użyciem buforów zarządzanych przez P4 lub zarządzanych przez użytkownika. Przesyłanie nie jest podzielone tak jak w wypadku PVM na trzy etapy, przygotowanie bufora, spakowanie i wysłanie. Realizuje się to wszystko przy użyciu jednej funkcji (chyba, że sami chcemy się zająć przygotowaniem bufora, wtedy musimy jeszcze wywołać funkcję *p4_msg_alloc()*). Słabością systemu w porównaniu z PVM'em lub Lindą może się wydawać brak możliwości tworzenia i wysyłania komunikatów złożonych z różnych typów danych przy korzystaniu z kodowania XDR, co będziemy mogli zaobserwować przy omawianiu poszczególnych funkcji.

1.1.1.1. Pakowanie i wysyłanie komunikatów

Do wysyłania komunikatów w systemie P4 mamy do dyspozycji następujące funkcje:

```
p4_send(int type, int to, int msg, int len)
```

```
p4_sendr(int type, int to, int msg,int len)
p4_sendx(int type, int to, int msg,int len,
int datatype)
p4_sendrx(int type, int to, int msg, int len,
int datatype)
p4_sendb(int type, int to,char *msg, int len)
p4_sendbr(int type, int to,char *msg, int len)
p4_sendbx(int type, int to,char *msg, int len,
int datatype)
p4_sendbrx(int type, int to,char *msg, int
len, int datatype)
```

Parametr *type* jest dowolną wybraną przez użytkownika liczbą całkowitą określającą typ komunikatu (odpowiednik *msgtag* w systemie PVM), służy do nadania pewnego rodzaju etykiety komunikatowi. *to* jest identyfikatorem procesu w systemie P4, który ma odebrać wiadomość. *len* określa w bajtach rozmiar wiadomości, która ma być przesłana. Procedury, które mają w swojej nazwie „b", tak jak *p4_sendb()*, *p4_sendbr()* i *p4_sendbrx()* zakładają, że komunikat *msg* znajduje się w buforze stworzonym wcześniej przez użytkownika poleceniem *p4_msg_alloc()*. Procedury z literą „r" w nazwie nie wracają (blokują proces) dopóki proces *to* nie potwierdzi odbioru. Litera „x" oznacza z kolei, że chcemy używać kodowania XDR. Przy wyborze tego rodzaju kodowania potrzebny jest dodatkowy parametr, który określa typ kodowanych danych *datatype*, może on mieć różne wartości (patrz Tabela 2).

Odbieranie komunikatów

Podobne jak w systemie PVM w P4 istnieje funkcja sprawdzająca czy nadszedł komunikat określonego typu lub od określonego nadawcy, takim odpowiednikiem funkcji *pvm_probe()* jest w systemie P4 funkcja,

```
int p4_message_available( int *req_type, int
*req_from)
```

która działa na bardzo podobnej zasadzie, wartość *req_type* określa typ komunikatu jakiego oczekujemy, a *req_from* od jakiego proces ma przyjść oczekiwany komunikat. Wartość –1 w dowolny z tych pól oznacza, że kryterium nie ma być brane pod uwagę. Do obierania komunikatów służy poniższa blokująca funkcja:

```
int p4_recv( int req_type, int req_from, char
**msg, int
        *len_rcvd )
```

Procedura ta pobiera pięć argumentów, znaczenie pierwszych dwóch zostało wyjaśnione przy okazji omawiania *p4_message_available()*, *msg* jest wskaźnikiem na wskaźnik, na dane typu char (znaki). Jeśli wartość tego pola wynosi NULL, system sam alokuje bufor zależnie od długości komunikatu. Jeżeli ta wartość nie jest równa NULL, wtedy użytkownik jest zobowiązany dostarczyć bufor poprzez wywołanie procedury *p4_msg_alloc()*. Ważną sprawą jest aby użytkownik zwolnił zaalokowany obszar, jeśli bufor nie ma być używany wielokrotnie. Do alokacji i zwalniania bufora służy para instrukcji:

```
char *pr_msg_alloc( int len )
void p4_msg_free( char *m )
```

Tak może wyglądać przykładowy program tworzący kilka procesów zdalnych, które wysyłają i odbierają po jednym komunikacie od każdego:

```
#include "p4.h"
main(int argc, char **argv)
{
p4_initenv(&argc, argv);
p4_create_procgroup();
worker();
p4_wait_for_end();
}
worker()
{
char *incoming, *msg = "hello";
int myid, size, nprocs, from, i, type;
myid = p4_get_my_id();
nprocs = p4_num_total_ids();
for (i=0; i < nprocs; i++)
{
if (i != myid)
p4_send(100, i, msg, strlen(msg)+1);
}
for (i=0; i < nprocs - 1; i++)
{
type = -1;
from = -1;
incoming = NULL;
p4_recv(&type, &from, &incoming, &size);
printf("%d received msg=:%s: from %d",myid,
incoming, from);
p4_msg_free(incoming);
}
}
```

Globalne operacje w systemie P4

Podobnie jak w przednich przypadkach P4 nie dorównuje możliwościami takim systemom jak PVM czy MPI. W P4 mamy do czynienia bardziej z operacjami globalnymi (umożliwiającymi komunikację z wszystkimi procesami na raz), niż z operacjami na grupach procesów w rozumieniu takim ja miało to miejsce w przypadku PVM i MPI. W P4 jako grupa traktowane są wszystkie procesy inicjowane i umieszczone we wcześniej opisywanym pliku *procgroup*. Nie ma możliwości dynamicznego tworzenia nowych grup i przegrupowywania procesów.

Możliwe jest rozgłaszanie broadcastowe komunikatów poprzez użycie funkcji:

```
p4_broadcast( int type, char *data, int data_len)
p4_broadcastx( int type, char *data, int data_len,
        int data_type)
```

Różnica pomiędzy tymi dwiema funkcjami polega na tym, że druga dokonuje kodowania XDR. Parametry podobne jak w przypadku funkcji *p4_send()*, etykieta komunikatu *type*, parametru *to* nie ma, gdyż komunikat będzie rozesłany do wszystkich procesów, *data*

zawiera treść komunikatu, *data_len* wyznacza jego długość, a dodatkowy parametr w *p4_broadcastx()* determinuje jakiego typu dane ma zakodować komunikat. Komunikat wysłany do wszystkich procesów może być odebrany jak każdy inny procedurą *p4_recv()*.

W P4 istnieje odpowiednik operacji redukcji istniejących w PVM i MPI. Funkcja realizująca coś takiego wygląda następująco:

```
p4_global_op( int type, char x, int nelem, int size, int
  (*op) (), int data_type )
```

data_type może przyjmować jedną z wartości P4INT, P4LNG, P4FLT lub P4DBL, a *op* może być funkcją zdefiniowaną przez użytkownika lub jedną z poniższych:

```
p4_int_absmax_op()
p4_int_absmin_op()
p4_int_max_op()
p4_int_min_op()
p4_int_mult_op()
p4_int_sum_op()
p4_dbl_absmax_op()
p4_dbl_absmin_op()
p4_dbl_max_op()
p4_dbl_min_op()
p4_dbl_mult_op()
p4_dbl_sum_op()
p4_flt_absmax_op()
p4_flt_absmin_op()
p4_flt_max_op()
p4_flt_min_op()
p4_flt_mult_op()
p4_flt_sum_op()
```

Nazwy tych funkcji są na tyle oczywiste, że nie ma potrzeby tłumaczyć jakie operacje arytmetyczne wykonują.

W P4 istnieje również możliwość synchronizacji barierowej, tak jak we wszystkich wcześniej przedstawionych narzędziach. Konieczność stosowania bariery może wynikać z sytuacji, kiedy po rozesłaniu wszystkim procesom danych do obliczeń, możemy kontynuować dalej pracę, gdy wszystkie odeślą swoje wyniki z powrotem. Bariera w P4 jest zaimplementowana w następujący sposób:

```
void p4_global_barrier( int type)
```

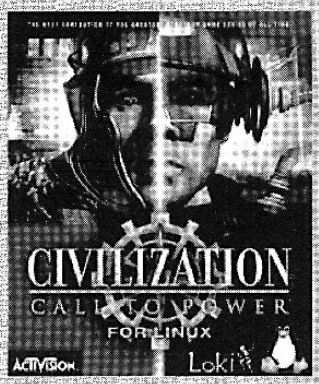
Jedynym parametrem tej procedury jest typ komunikatu *type*, który jest użyty do wewnętrznego przesyłania komunikatów. Powoduje ona, że wszystkie wywołujące ją procesy zawieszają się do czasu, aż wszystkie procesy wyspecyfikowane w pliku *procgroup* nie wywołają tej procedury.

Więcej szczegółowych informacji na temat P4 oraz możliwościach tego pakietu związanych z programowaniem w systemach z pamięcią dzieloną można przeczytać w internetowej publikacji R. Butler, E. Lusk „User's Guide to the p4 Parallel Programming System", The University of North Florida, Department of Computer Science, Jackonville, Florida, 1992. ■

Civilization: Call To Power na Linuxa

Najpopularniejsza gra strategiczna wszechczasów. Dotychczas sprzedano 2,5 miliona kopii gier z tej serii.

*PLN 206,-



- zwiększona liczba jednostek, budynków, cudów i technologii,
- nowe ogromne królestwa - kosmos i morze,
- w pełni animowana 16-bitowa grafika wysokiej rozdzielczości,
- nowe strategie, między innymi niekonwencjonalne sposoby prowadzenia wojen,
- rozszerzone opcje dyplomatyczne i handlowe,
- oprogramowanie do tworzenia scenerii,
- możliwość gry wieloosobowej poprzez sieci lokalne i Internet.

TAO SYSTEMS POLSKA

TAO Systems Polska s.c.
ul. Poznańska 21/26
00-685 Warszawa
tel./fax (22) 628-77-02
e-mail: tao@tao.com.pl
WWW: http://www.tao.com.pl/

*Cena bez podatku VAT