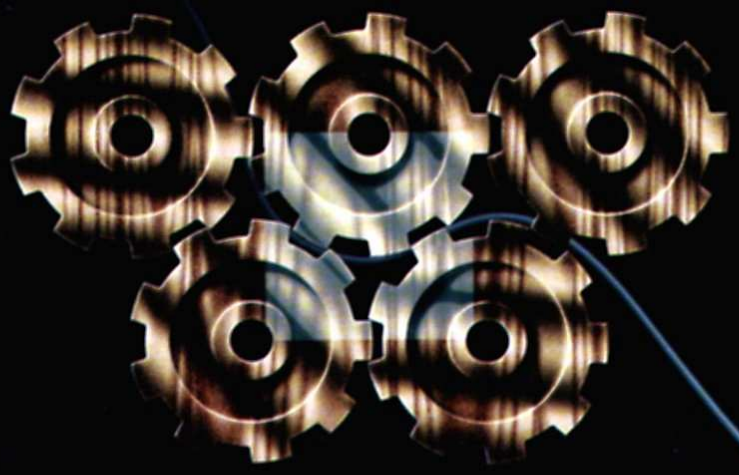
CD-ROM: SCO OpenServer 5.0.5, KEA! X v4.1

11/99

magazyn

Linux & UNIX

Programowanie w języku PostScript



PHP, procesy i PDF

TCP/IP od podstaw

NDS for Solaris 2.0

Pakiety RPM

Technologia Storage Area Network

Pakiet PVM

CD ROM

NR 11(15)/99 LISTOPAD CENA 17 zł ISSN 1506-1299

magazyn Linux & UNIX

Adres kontaktowy: magazyn@tao.com.pl

Hacking In Poland '99 6
Wojciech Bojdoł

Programowanie w języku PostScript II 7
Adam Dawidziuk

E-commerce w świecie Unixa na przykładzie InterMarket 3D (część III) 10

Produkty

InterSystems Caché. Nowa technologia obiektowej bazy danych wychodzi naprzeciw złożonym wymaganiom biznesu 12
Kevin Dick

KEA! X v4.1 18

Programowanie

Upiększanie aplikacji 20
Marek Sawerwain

PHP, procesy i PDF 23
Piotr Bolek

Technologie Sieciowe

PVM - Komunikacja 26
Piotr Koch

Technologia Storage Area Network 30
Marcin Kabaj

TCP/IP od podstaw - część I 32
Wojciech Bojdoł

Wprowadzenie do NDS for Solaris 2.0 35

UNIX

Instalacja SCO OpenServer Release 5.0.5 41
Andrzej Boczar

Linux dla każdego

Filozofia RPMa 44
Michał Kuratczyk

Wdrożenia

FreeBSD wśród NT, czyli natd w praktyce 49
Marcin Oczkowski

Uniwersytet Warmińsko-Mazurski w Olsztynie 51
Maciej Orzechowski

magazyn Linux & UNIX

PVM - Komunikacja

Piotr Koch

Komunikaty i funkcje z nimi związane są podstawą działania większości systemów rozproszonych, a w szczególności tych opartych na modelu proceduralnego przesyłania komunikatów. Komunikaty są podstawową metodą wymiany informacji między procesami.

Przesyłanie komunikatu w PVM składa się z trzech kroków. Pierwszy to zainicjowanie bufora nadawczego przez wywołanie `pvm_initsend()` lub `pvm_mkbuf()`. Drugi polega na tym, że komunikat musi być "spakowany" z użyciem różnych kombinacji funkcji `pvm_pk...()`, W trzecim kompletny komunikat jest wysyłany do innego procesu przez wywołanie `pvm_send()` lub rozesłany przy użyciu funkcji `pvm_mcast()`.

Komunikat odebrany przez inny proces, blokującą lub nie blokującą funkcją służącą do odbierania komunikatów, jest "rozpakowywany" z bufora odbiorczego. W funkcjach odbierających można wyspecyfikować czy komunikaty mają być odbierane od wszystkich procesów, czy od konkretnego źródła, czy oznaczone specjalną etykietą (ang. *message tag*), czy może jakąś kombinacją wszystkich wcześniej wymienionych warunków. Istnieje też specjalna funkcja, która sprawdza jakie wiadomości nadchodzą bez ich odbierania.

Bufory komunikatów

Bufory komunikatów są jednym z podstawowych pojęć związanych z przesyłaniem komunikatów w PVM. Każdy komunikat musi przed wysłaniem zostać odpowiednio przygotowany i umieszczony w buforze (spakowany).
Przed każdym spakowaniem i wysłaniem komunikatu bufor musi być odpowiednio przygotowany, co może się odbyć przez wywołanie funkcji:

```
int bufid = pvm_initsend( int encoding )
```

Jeśli użytkownik używa tylko jednego bufora nadawczego (a jest to typowe użycie), wtedy funkcja `pvm_initsend()` jest jedyną potrzebną funkcją operującą na buforach. Procedura ta czyści bufor nowy, do którego może być spakowany nowy komunikat. Sposób w jaki są kodowane dane w buforze jest określony przez parametr `encoding`, a nowy identyfikator bufora jest podstawiany pod zmienną `bufid`. Istnieją trzy sposoby kodowania:

PvmDataDefault kodowanie XDR (ang. *eXternal Data Representation*), internetowy standard kodowania danych, konwersja do tego formatu pozwala na komunikację między hostami o różnej reprezentacji danych, co pozwala to na rozproszone przetwarzanie w sieciach heterogenicznych. Jeśli użytkownik ma pewność, że w przetwarzaniu nie biorą udziału maszyny o różnej reprezentacji danych i ma pewność, że będą rozumieć przesyłane między sobą dane, w swoich wewnętrznych formatach może użyć kodowania *PvmDataRaw*, co zmniejsza o wiele koszty przesyłania tych danych.
PvmDataRaw dane nie są kodowane. Komunikaty są wysyłane w oryginalnej formie. Jeśli odbiorca tak zakodowanego komunikatu nie może zrozumieć danych, występuje błąd podczas rozpakowywania.
PvmDataInPlace dane pozostają w miejscu, oszczędza się na kosztach związanych z pakowaniem. Bufor zawiera jedynie rozmiar i wskaźnik na dane do wysłania. Kiedy zostaje wywołany *pvm_send()*, dane są kopiowane bezpośrednio z pamięci użytkownika. Opcja redukuje do minimum ilość kopii danej do wysłania, ale użytkownik nie może modyfikować wartości wysyłanej zmiennej w czasie miedzy pakowaniem a wysyłaniem komunikatu.

Dalej opisywane procedury operujące na buforach są potrzebne tylko wtedy, gdy użytkownik chce zarządzać wieloma buforami komunikatów wewnątrz aplikacji. W większości aplikacji z przesyłaniem komunikatów nie ma takiej potrzeby. W systemie PVM standardowo istnieje jeden aktywny bufor nadawczy na każdy proces. Programista może utworzyć dowolną liczbę buforów komunikatów i przełączać je między pakowaniem a wysyłaniem. Procedury do pakowania, wysyłania i odbierania działają tylko na aktywnych buforach.

Gdy użytkownik odważy się na bardziej zaawansowaną obsługę buforów może wykorzystać następujące funkcje, które tworzą nowe bufory, ustawiają je jako aktywne w celu użycia do odbioru lub nadania komunikatu i likwidują je gdy nie są już potrzebne.

```
int bufid = pvm_mkbuf( int encoding )
```

Procedura ta `pvm_mkbuf()` tworzy nowy bufor nadawczy i specyfikuje metodę kodowania używaną do pakowania komunikatów. Zwraca identyfikator bufora `bufid`.

```
int info = pvm_freebuf( int bufid )
```

Wywołanie tej konstrukcji powoduje zlikwidowanie bufora o identyfikatorze `bufid`. Powinno to być zrobione po tym jak komunikat zostanie wysłany i nie jest już potrzebny. Gdy chcemy wysłać nową wiadomość musimy stworzyć nowy bufor wywołując `pvm_mkbuf()`. Nie ma potrzeby używania tych funkcji jeśli korzysta się z funkcji `pvm_initsend()`, która jest polecaną dla użytkowników.

```
int bufid = pvm_getsbuf( void )
int bufid = pvm_getrbuf( void )
```

Do stwierdzenia, który bufor nadawczy jest aktywny służy funkcja `pvm_getsbuf()` zwracająca identyfikator tego bufora. Funkcja `pvm_getrbuf()` robi to samo dla bufora odbiorczego.

```
int oldbuf = pvm_setsbuf( int bufid )
int oldbuf = pvm_setrbuf( int bufid )
```

Tymi procedurami można wpływać na to, które bufory mają być aktywnymi buforami nadawczymi lub odbiorczymi, z zachowaniem stanu poprzednich buforów i ich identyfikatorów. Dzięki tym procedurom możliwe jest przesłanie komunikatu bez jego przepakowywania, co ilustruje poniższy przykład:

```
bufid = pvm_recv( src, tag );
oldid = pvm_setsbuf( bufid );
info = pvm_send( dst,tag );
info = pvm_freebuf( oldid );
```

Pakowanie danych

Dane przed wysłaniem z procesu do procesu muszą być odpowiednio przygotowane i umieszczone w buforze nadawczym. Proces ten nazywamy w systemie PVM pakowaniem.

Każda z poniżej przedstawionych procedur pakuje tablice danych określonego typu do aktywnego buforu nadawczego. Mogą być one wywoływane wielokrotnie w celu spakowania danych do jednego komunikatu, komunikat może zawierać kilka tablic różnego typu. Struktury C mogą być przesyłane tylko poprzez indywidualne pakowanie poszczególnych elementów. Nie ma ograniczeń co do stopnia skomplikowania pakowanych komunikatów, ale aplikacja powinna rozpakowywać dokładnie w ten sam sposób w jaki był pakowany. Chociaż nie ma co do tego silnych restrykcji, jednak bez przestrzegania tej reguły nie jest możliwe stworzenie bezpiecznej i poprawnie działającej aplikacji.

Argumentami dla każdej instrukcji pakowania są: wskaźnik na pierwszy element do spakowania, liczba `nitem`,określająca całkowitą ilość elementów w pakowanej tablicy i `stride` wyznaczający z jakim krokiem odbywa się pakowanie. Gdy `stride` wynosi 1 pakowana jest cała tablica, gdy wynosi 2 to pakowany jest co drugi element tablicy i tak dalej. Wyjątek stanowi `pvm_pkstr()`, który już z definicji pakuje ciągi znaków zakończone znakiem NULL, co za tym idzie nie potrzebuje ani wartości `nitem` ani `stride`. Zbiór funkcji do pakowania tablic zawiera Tabela 1. Specyficznym przypadkiem instrukcji pakującej jest polecenie:

```
int info = pvm_packf ( const )
```

Sposób użycia tej funkcji jest podobny jak standardowej funkcji `printf()`, formatujące wyrażenia specyfikują co i jak ma być pakowane do bufora nadawczego.

Tabela 1.

```
int info = pvm_pkbyte ( char *cp, int nitem, int stride )
int info = pvm_pkcplx ( float *xp, int nitem, int stride )
int info = pvm_pkdcplx ( double *zp, int nitem, int stride )
int info = pvm_pkdouble ( double *dp, int nitem, int stride )
int info = pvm_pkfloat ( float *fp, int nitem, int stride )
int info = pvm_pkint ( int *np, int nitem, int stride )
int info = pvm_pklong ( long *np, int nitem, int stride )
int info = pvm_pkshort ( short *np, int nitem, int stride )
int info = pvm_pkstr ( char *cp )
```

Wysyłanie i odbieranie danych

Podstawowym zbiorem systemów opartych na przesyłaniu komunikatów są funkcje realizujące komunikację pomiędzy procesami, czyli procedury wysyłające i odbierające komunikaty. System zapewnia dużą różnorodność tych funkcji od blokujących do nie blokujących, umożliwiających pakowanie przed wysłaniem i realizujące multicasting. Procedury komunikacyjne działają asynchronicznie, co nie jest bez znaczenia dla wydajności aplikacji tworzonych w PVM'ie. Funkcje realizujące wysyłanie komunikatów `topvm_send`, `pvm_mcast` i `pvm_psend`.

```
int info = pvm_send( int tid, int msgtag )
```

pvm_send() wysyła komunikat z aktywnego bufora nadawczego do procesu PVM identyfikowanego przez `tid`. `msgtag` służy do nadawania etykiet treści komunikatów. Procedyra `pvm_send()` jest asynchroniczna. Przetwarzanie w procesorze jest kontynuowane wkrótce po tym jak upewni się on, że komunikat będzie odebrany przez adresata. Jest to przeciwieństwo do komunikacji synchronicznej, w której obliczenia na wysyłającym procesorze są wstrzymywane na czas odbioru komunikatu przez drugi procesor. `pvm_send()` najpierw sprawdza czy odbiorca znajduje się na tym samym komputerze. Jeśli tak i host ten jest multiprocesorem, wtedy wywoływane są odpowiednie procedury przenoszące dane miedzy procesami.

```
int info = pvm_mcast( int *tids, int ntask,
        → int msgtag )
```

pvm_mcast() realizuje rozsyłanie komunikatu zgromadzonego w aktywnym buforze nadawczym do `ntask` zadań, których identyfikatory znajdują się w tablicy `tids`. Komunikat nie jest przesyłany do procesu wywołującego, nawet jeśli jego TID znajduje się w tablicy `tids`. Podobnie jak `pvm_send()`, instrukcja ta działa w sposób asynchroniczny. Komunikat może być odebrany przez proces odbiorczy poprzez wywołanie funkcji `pvm_recv()` lub `pvm_nrecv()`.

```
int info = pvm_psend( int tid, int msgtag,
        → void *vp, int cnt, int type )
```

`pvm_psend()` to kolejna procedura umożliwiająca wysyłanie komunikatu. W odróżnieniu od poprzednich funk-

cji nie wymaga aby komunikat czekał spakowany w buforze nadawczym. Dane wskazywane przez `vp` są automatycznie pakowane, ilość tych danych określa `cnt`, a typ specyfikuje zmienna `type`, która może przybierać jedną z poniższych wartości:

Definicje typów danych funkcji pvm_psend().

PVM_STR	PVM_FLOAT
PVM_BYTE	PVM_CPLX
PVM_SHORT	PVM_DOUBLE
PVM_INT	PVM_DCPLX
PVM_LONG	PVM_UINT
PVM_USHORT	PVM_ULONG

Odbieranie komunikatów

Można by filozoficznie stwierdzić co zostało wysłane powinno być odebrane. Tak więc oprócz funkcji wysyłających komunikaty w PVM istnieją też funkcje odpierające je. PVM posiada kilka metod odbierania komunikatów. Nie istotne jest jakim poleceniem został wysłany komunikat, odebranie może nastąpić dowolną funkcją.

```
int bufid = pvm_recv( int tid, int msgtag )
```

`pvm_recv()` to blokująca wersja procedury odbierającej, która czeka aż komunikat oznaczony etykietą `msgtag` od procesu `tid` nadejdzie do lokalnego demona `pvmd`. Gdy `msgtag` lub `tid` mają wartość –1, to kryteria te nie są brane pod uwagę podczas odbioru komunikatu. PVM gwarantuje kolejność przychodzących komunikatów.

```
int bufid = pvm_nrecv( int tid, int msgtag )
```

`pvm_nrecv()` to nie blokująca funkcja do odbierania komunikatów. Jeśli komunikat jakiego żądamy jeszcze nie dotarł, program nie jest zawieszany jak w przypadku `pvm_recv`, ale następuje powrót i funkcja zwraca wartość `bufid` = 0. Procedura może być wywoływana wielokrotnie w celu sprawdzania czy nie przybył oczekiwany komunikat, a w międzyczasie program może wykonywać jakąś inną pracę. Wykorzystanie tej funkcji może podnieść znacznie efektywność pisanego programu. Gdy wiadomość zostaje odebrana, jest umieszczana w nowym aktywnym buforze, który jest dla niej tworzony, a funkcja zwraca jego identyfikator `bufid`.

```
int bufid = pvm_probe( int tid, int
→           msgtag )
```

`pvm_probe()` służy to sprawdzenia, czy nadszedł jakiś komunikat. W przypadku gdy jest już komunikat nadający się do odebrania, funkcja zwraca identyfikator bufora, a w przeciwnym wypadku zawracana jest wartość 0. Poprzez użycie funkcji `pvm_bufinfo()` możliwe jest poznanie większej ilości szczegółów na temat komunikatu znajdującego się w buforze, jeszcze przed jego odebraniem.

```
int bufid = pvm_trecv( int tid, int msgtag,
→           struct timeval *tmount )
```

`pvm_trecv()` wypełnia lukę pomiędzy blokującym `pvm_recv()` a nie blokującym `pvm_nrecv()`. Funkcja ta pozwala określić jak długo proces ma być zawieszany w oczekiwaniu na nadejście komunikatu. Podanie dużej wartości `timeval` powoduje działanie tej funkcji zbliżone do `pvm_recv()`, a wartość równa zero zamienia tą funkcję w nie blokujący `pvm_nrecv()`. Posługiwanie się tą funkcją zamiast `pvm_recv()` pozwala zabezpieczyć się przed zawieszeniem procesu w oczekiwaniu na komunikat, który nie doszedł (z powodu błędu systemu lub aplikacji).

```
int info = pvm_bufinfo( int bufid, int *bytes,
→          int *msgtag, int *tid )
```

`pvm_bufinfo()` pozwala uzyskać etykietę komunikatu `msgtag`, identyfikator `tid` procesu, który go nadesłał i jego rozmiar w bajtach.

```
int info = pvm_precv( int tid, int msgtag,
→  void *vp, int cnt, int type, int *rtid, int
→  *rtag, int *rcnt )
```

`pvm_precv()` to funkcja odwrotna do `pvm_psend()`, która łączy w sobie blokującą funkcje odbierającą z rozpakowującą komunikat do bufora.

Rozpakowywanie danych

Odebrane wiadomości (komunikaty) należy rozpakować, to znaczy przekształcić z powrotem do takiej postaci jak przed spakowaniem (taka sama ilość elementów w tablicy, taki sam typ danych). Jeśli używamy kodowania XDR, przekształcenie odbywa się na reprezentacje binarne typów charakterystyczne dla danej maszyny. Rozpakowanie komunikatu powinno odbywać się w ten sam sposób co jego pakowanie, nie jest to rygorystycznie przestrzegane przez system, ale nie przestrzeganie tej zasady może prowadzić do powstawania błędów trudnych do wykrycia. Parametry funkcji rozpakowujących mają takie samo znaczenie jak w przypadku funkcji pakujących. Do dyspozycji mamy funkcje znajdujące się w Tabeli 2.

Podobnie też jak w przypadku funkcji pakujących

Tabela 2.

```
int info = pvm_upkbyte( char *cp, int nitem, int stride )
int info = pvm_upkcplx ( float *xp, int nitem, int stride )
int info = pvm_upkdcplx ( double *zp, int nitem, int stride )
int info = pvm_upkdouble ( double *dp, int nitem, int stride )
int info = pvm_upkfloat ( float *fp, int nitem, int stride )
int info = pvm_upkint ( int *np, int nitem, int stride )
int info = pvm_upklong ( long *np, int nitem, int stride )
int info = pvm_upkshort ( short *np, int nitem, int stride )
int info = pvm_upkstr ( char *cp )
```

istnieje uniwersalna funkcja rozpakowująca działająca na zasadzie podobnej do `printf`:

```
int info = pvm_unpack( const char *fmt, ...)
```

Przetwarzanie w grupach procesów

Możliwość podziału procesów na grupy jest bardzo istotną cechą systemów rozproszonych. Program rozproszony zazwyczaj składa się z jednego programu wczytującego dane, rozdzielającego zadania i odbierającego wyniki, oraz wielu innych programów uruchamianych jako procesy zdalne realizujące jakieś określone operacje. Bardzo wygodne jest pogrupowanie procesów realizujących takie same zadania w grupy. Grupowanie takie daje wiele nowych możliwości. Możliwe jest stworzenie synchronizacji barierowej, przesyłanie komunikatów zwane broadcastowym oraz wykonywanie odwrotnej do tej operacji redukcji.

W systemie PVM istnieje możliwość tworzenia dynamicznych grup procesów. Grupa jest tworzona w momencie kiedy jest do niej dołączany pierwszy proces. Procesy mogą dołączać się i opuszczać grupę w dowolnej chwili bez powiadamiania innych procesów. Broadcastowe komunikaty mogą być wysyłane przez procesy nie będące członkami danej grupy. Inaczej mówiąc funkcje grupowe mogą być wywoływane przez procesy nie będące członkami żadnych grup, wyjątkiem oczywiście są operacje `pvm_lvgroup()`, `pvm_barrier()` i `pvm_reduce()`, których natura wymaga aby proces wywołujący był członkiem grupy.

Funkcje do dynamicznych grup procesów są nadbudowane nad rdzennymi procedurami PVM. Jeśli chcemy wykorzystywać te procedury, musimy program dodatkowo połączyć z biblioteką libgpvm3.a. Demon `pvmd` nie wykonuje funkcji do dynamicznego grupowania procesów. Zadanie to należy do serwera grup, który automatycznie startuje, gdy po raz pierwszy zostaje użyta funkcja do przetwarzania w grupach.

Żeby było możliwe tworzenie grup procesów, muszą istnieć funkcje pozwalające procesom tworzyć grupę, przyłączać się do niej i opuścić ją jeśli jest to potrzebne. Operacje te realizują dwie poniższe funkcje:

```
int inum = pvm_joingroup( char *group )
int info = pvm_lvgroup( char *group )
```

Pierwsze wywołanie `pvm_joingroup()` tworzy grupę o nazwie `group` i dołącza do niej proces wywołujący. Każde kolejne wywołanie tej funkcji z nazwą tej samej grupy powoduję dodanie do niej procesu wywołującego. Zwracaną wartością przez tę funkcję jest numer procesu w grupie. Numer ten jest przydzielany na zasadzie pierwszy wolny, nie ma pewności, że kolejny będzie większy od poprzedniego, ani że proces który opuścił wcześniej grupę, ponownie się przyłączając otrzyma ten sam. Numer ten może mieć wartość od 0 do ilość członków grupy minus jeden. Zadania mogą należeć do wielu grup. `Pvm_lvgroup()` służy do opuszczania grupy przez proces.

Istnieje cały szereg użytecznych funkcji pozwalających na podstawie jednych informacji o procesie, uzyskiwać inne.

Poniższa funkcja pozwala uzyskać TID procesu na podstawie nazwy grupy, do której należy i jego numeru w grupie.

```
int tid = pvm_gettid( char *group, int inum )
```

Działanie funkcji `pvm_getinst()` umożliwia otrzymanie numeru w grupie procesu, który do niej należy na podstawie TID i nazwy grupy.

```
int inum = pvm_getinst( char *group, int tid )
```

W ustaleniu rozmiaru grupy pomocna może być funkcja `pvm_gsize()`.

```
int size = pvm_gsize( char *group )
```

W przypadku kiedy korzystamy z grup procesów możliwe jest zastosowanie, wcześniej już opisywanego, mechanizmu synchronizacji barierowej. Funkcją realizującą mechanizm bariery jest `pvm_barrier()`.

```
int info = pvm_barrier( char *group, int count )
```

Wywołując tę funkcję proces jest blokowany na czas, kiedy `count` procesów grupy nie wywoła polecenia `pvm_barrier()`. Generalizując `count` powinno być całkowitą liczbą procesów w grupie. Jednak wymagane jest wyspecyfikowanie tego parametru, ponieważ grupy procesów są dynamiczne, a liczba członków może się zmieniać na bieżąco. Błędne określenie liczby procesów, które mają dotrzeć do blokady w celu synchronizacji może doprowadzić do blokady.

Dodatkową zaletą komunikacji, gdy mamy do czynienia z grupami, jest możliwość adresowania komunikatów wyłącznie do członków grupy. Komunikację taką zwykło się nazywać rozgłaszaniem broadcastowym. Gdy w poniższym poleceniu jako parametru `msgtag`, użyjemy adresowania do wszystkich zadań, komunikat otrzymają wszyscy członkowie grupy `group`.

```
int info = pvm_bcast( char *group, int msgtag )
```

Kolejną bardzo ciekawą funkcją realizującą komunikację w PVM jest polecenie `pvm_reduce()`, realizujące operację redukcji.

```
int info = pvm_reduce( void (*fun)(),
→          void *data, int nitem, int datatype,
→          int msgtag, char *group, int root )
```

`pvm_reduce()` wykonuje arytmetyczną operacje przez wszystkich członków grupy. Wynik tej operacji jest dostępny dla procesu zwanego korzeniem. Istnieje kilka zdefiniowanych operacji, które są wykonywane na członkach grupy (PvmMax, PvmMin, PvmSum, PvmProduct), ale użytkownik może też sam zdefiniować operacje redukcji (parametr `fun`). Instrukcja ta nie jest blokująca. Jeśli proces wykona to polecenie i opuści grupę za nim proces korzeń (root) sfinalizuje redukcję wystąpi błąd. ■