Pierwsze kroki w środowisku PVM

Zakres ćwiczenia

W tym ćwiczeniu dowiesz się, w jaki sposób utworzyć maszynę wirtualną, jak napisać swój pierwszy program wykorzystujący środowisko PVM i jak taki program uruchomić.

Etapy tworzenia i uruchamiania aplikacji równoległej w środowisku PVM

W celu uruchomienia przetwarzania pod kontrolą środowiska PVM niezbędne jest podjęcie następujących kroków:

A. Przygotowanie programów:

1. Przygotowanie kodów źródłowych w języku C lub Fortran.

   W kodzie źródłowym programu można wykorzystać funkcje z biblioteki PVM, do których interfejs dla języka C znajduje się w pliku nagłówkowym `pvm3.h`. Opis podstawowych funkcji z tej biblioteki znajduje się w dostarczonych materiałach dodatkowych.

2. Kompilacja kodów źródłowych i konsolidacja z odpowiednimi bibliotekami.

   W przypadku języka C należy dołączyć bibliotekę `libpvm3.a`, oraz dodatkowo `libgpvm3.a`, jeśli wykorzystywane są funkcje grupowe. Dla języka Fortran jest jedna biblioteka `libfpvm3.a`. Przykładowa kompilacja pliku o nazwie `przyklad_pvm.c` wyglądałaby zatem następująco:

   ```
   cc przyklad_pvm.c -lpvm3.a -lgpvm3.a
   ```

   Ponieważ PVM umożliwia skonfigurowanie maszyny wirtualnej w środowisku heterogenicznym (dopuszcza maszyny o różnych architekturach sprzętowych) konieczne jest przygotowanie i właściwe rozmieszczenie plików binarnych w systemach plików poszczególnych węzłów lub w odpowiednich podkatalogach jeśli węzły współdzielą ten sam system plików.

B. Skonfigurowanie maszyny wirtualnej:

1. Wybranie odpowiednich węzłów i ewentualnie przygotowanie pliku z ich opisem.

   Węzły należy dobrać stosownie do potrzeb poszczególnych programów tworzących aplikację równoległą. W każdym z węzłów musi być dostępne konto użytkownika, na które można się zalogować w celu uruchomienia demona i ewentualnie w celu skompilowania kodów źródłowych. Konfiguracja maszyny wirtualnej może odbywać się na dwa sposoby: pierwszy z uruchamianych demonów powinien dostać plik z opisem węzłów, lub też, w przypadku braku takiego pliku, można kolejne węzły maszyny wirtualnej dodawać ręcznie za pomocą poleceń konsoli pvm (omówionych w dalszej części materiałów).

2. Uruchomienie demonów na poszczególnych węzłach.

   Demon na pierwszym węźle maszyny wirtualnej uruchamiany jest przez podanie jego nazwy na terminalu lub przez uruchomienie konsoli PVM. Jego rola w systemie jest szczególna, gdyż odpowiedzialny jest za zmianę konfiguracji maszyny wirtualnej. Demon ten musi zatem działać do końca pracy maszyny wirtualnej, co oznacza, że nie można dynamicznie usunąć pierwszego węzła tworzącego daną konfigurację. Pierwszy demon może zostać uruchomiony z nazwą plik zawierającego opis węzłów. W zależności od opisu danego węzła demon jest na nim uruchamiany natychmiast i

węzeł wchodzi do konfiguracji maszyny wirtualnej lub zapamiętywane są tylko parametry węzła, a jego dołączenie i tym samym uruchomienie demona odkładane jest na później.

3. Ewentualna dynamiczna zmiana konfiguracji początkowej.

   Początkowa konfiguracja maszyny wirtualnej może ulec zmianie w wyniku wywołania odpowiednich funkcji PVM przez jedno z zadań lub w wyniku wydania odpowiedniego polecenia na konsoli PVM. Jeżeli konieczne jest ustawienie określonych parametrów pracy węzła, należy to zrobić w pliku opisu węzłów przed uruchomieniem pierwszego demona.

C. Uruchomienie zadań:

Możliwe są trzy alternatywne sposoby uruchomienia zadań na maszynie wirtualnej:

1. Uruchomienie procesu na terminalu jednego z węzłów.

   Zadanie uruchamiane w ten sposób najpierw jest procesem w lokalnym systemie operacyjnym węzła, na terminalu którego wydawane jest odpowiednie polecenie. Następnie, przy wywołaniu pierwszej funkcji z podstawowej biblioteki PVM (nie może to być funkcja grupowa) następuje przyłączenie procesu do maszyny wirtualnej i dopiero wówczas staje się on zadaniem w sensie środowiska PVM.

2. Uruchomienie zadania poleceniem `spawn` z konsoli PVM .

   Zadanie uruchamiane w ten sposób również jest procesem w lokalnym systemie operacyjnym węzła, na którym zostało uruchomione, ale niemal natychmiast staje się też zadaniem PVM.

3. Uruchomienie zadania przez inne zadanie działające na maszynie wirtualnej

   Możliwe jest to dzięki funkcja `pvm_spawn`, dostępnej w bibliotece PVM. Można w ten sposób uzyskać hierarchię przodek-potomek podobną do hierarchii procesów np. w systemie UNIX. Różnica jest jednak taka, że w środowisku PVM w ogólności mamy do czynienia z lasem zadań w odróżnieniu od drzewa procesów w Uniksie, gdyż zadania uruchamiane w pierwszy z wymienionych sposobów nie mają przodka na maszynie wirtualnej. Hierarchia zadań na maszynie wirtualnej w żaden sposób nie odzwierciedla hierarchii w lokalnym systemie operacyjnym. Wszystkie zadania, uruchamiane w drugi lub trzeci sposób, jako procesy w lokalnym systemie operacyjnym są potomkami demona PVM, a zadania uruchamiane w pierwszy sposób są oczywiście potomkami powłoki, która zinterpretowała polecenie uruchomienia.

Podstawowe polecenia konsoli PVM

Uruchom środowisko PVM za pomocą wcześniej poznanego polecenia `pvm`. Pierwszym krokiem będzie uzyskanie pomocy na temat dostępnych komend konsoli PVM. Pomoc taką, jak łatwo się domyślić, można uzyskać za pomocą polecenia `help`:

```
pvm> help
help        Print helpful information about a command
Syntax:  help [ command ]
Commands are:
  add         Add hosts to virtual machine
  alias       Define/list command aliases
  conf        List virtual machine configuration
  delete      Delete hosts from virtual machine
  echo        Echo arguments
  export      Add environment variables to spawn export list
  getopt      Display PVM options for the console task
  halt        Stop pvmds
  help        Print helpful information about a command
  id          Print console task id
  jobs        Display list of running jobs
  kill        Terminate tasks
  mstat       Show status of hosts
  names       List message mailbox names
  ps          List tasks
  pstat       Show status of tasks
  put         Add entry to message mailbox
  quit        Exit console
  reset       Kill all tasks, delete leftover mboxes
  setenv      Display or set environment variables
  setopt      Set PVM options - for the console task *only*!
  sig         Send signal to task
  spawn       Spawn task
  trace       Set/display trace event mask
  unalias     Undefine command alias
  unexport    Remove environment variables from spawn export list
  version     Show libpvm version
```

Aby uzyskać pomoc na temat konkretnego polecenia, wystarczy wydać komendę `help <nazwa_polecenia>`, na przykład, aby uzyskać informację na temat wcześniej już poznanego polecenia `quit` należy napisać `help quit`:

```
pvm> help quit
quit        Exit console
Syntax:  quit
```

Spróbuj teraz uzyskać informację na temat polecenia `version`, a następnie wydać to polecenie:

```
pvm> help version
version     Show libpvm version
Syntax:  version
pvm> version
3.4.5
```

Wreszcie, uzyskaj informację na temat polecenia `conf` i wydaj to polecenie:

```
pvm> help conf
conf        List virtual machine configuration
Syntax:  conf
Output fields:
  HOST    host name
  DTID    tid base of pvmd
  ARCH    xhost architecture
  SPEED   host relative speed
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
                linuxhost    40000    LINUX    1000 0x00408841
pvm>
```

Jak widzisz, polecenie `conf` służy do pokazywania konfiguracji maszyny wirtualnej. Początkowo konfiguracja ta obejmuje tylko jeden węzeł – ten, na którym uruchomiono konsolę PVM. Kolejnym naszym krokiem będzie dodanie kilku dodatkowych węzłów, co – przynajmniej teoretycznie – pozwoli nam na uzyskanie większej mocy obliczeniowej.

Konfiguracja maszyny wirtualnej

Uruchom, jeżeli jeszcze tego nie zrobiłeś, konsolę PVM. Obecnie dodamy kolejne węzły do naszej maszyny wirtualnej. Przed kontynuowaniem ćwiczenia upewnij się, że posiadasz poprawnie skonfigurowaną sieć. Dołączenia nowego węzła obejmuje uruchomienie na nim zdalnie demona PVM. Upewnij się, że możesz wykonywać zdalne polecenia na maszynie, którą masz zamiar dołączyć do swojej maszyny wirtualnej. Jeżeli używasz `ssh`, może to wyglądać następująco:

```
user@linuxhost:~/pvm3/src> ssh sirius who
Password:
user   pts/1          Jul 20 09:44 (linuxhost.pl)
inf66226 pts/2        Jul 21 09:42 (150.254.149.161)
```

Do dodawania nowych węzłów służy polecenie `add` konsoli PVM. Dodanie węzłów o nazwie `lab-143-2`, `lab-143-4`, `lab-143-5` w najprostszym wypadku (jeżeli nazwa użytkownika używanego przez demona PVM ma być identyczna jak nazwa użytkownika na bieżącej maszynie) wyglądać będzie następująco:

```
user@linuxhost:~>pvm
pvm> add lab-143-2
Password:
1 successful
                    HOST     DTID
                lab-143-2    c0000
pvm> add lab-143-4
Password:
1 successful
                    HOST     DTID
                lab-143-4   100000
pvm> add lab-143-5
Password:
1 successful
                    HOST     DTID
                lab-143-5   140000
```

Przy używaniu `ssh` dodawanie każdego węzła wymagania podania hasła użytkownika używanego na dołączanym węźle; jeżeli więc używamy konta użytkownika `user` na maszynach `linuxhost` oraz

lab-143-2, to dołączając maszynę lab-143-2 należy podać hasło użytkownika user na maszynie lab-143-2. Podawanie hasła dla większej liczby maszyn może być dość męczące, należy wtedy zapoznać się z użytkowaniem programu ssh-agent.

Wydanie polecenia conf pokazuje, że obecnie maszyna wirtualna składa się z kilku węzłów:

```
pvm> conf
4 hosts, 2 data formats
                     HOST      DTID     ARCH   SPEED       DSIG
                linuxhost     40000    LINUX    1000 0x00408841
                lab-143-2     c0000  LINUX64    1000 0x00408c41
                lab-143-4    100000  LINUX64    1000 0x00408c41
                lab-143-5    140000  LINUX64    1000 0x00408c41
```

Można teraz wyjść z konsoli PVM za pomocą polecenia quit.

Alternatywny sposób polega na przygotowaniu wcześniej pliku konfiguracyjnego o dowolnej nazwie. Poniżej znajduje się przykładowy plik konfiguracyjny o nazwie machines.conf.

```
#komentarz
lab-143-2 sp=100

* sp=200 ep=/home/szopen/pvm3/bin/LINUX64 dx=/usr/bin/pvmd
&lab-143-6 lo=szopen so=pw ep=/home/szopen/pvm3/bin/LINUX
```

Jak widać, # oznacza początek komentarza. Używanych opcji nie można używać dla polecenia add. Ich znaczenie wyjaśnione jest w tabeli 1. Gwiazdka zamiast nazwy węzła oznacza wartości domyślne dla dalej dodawanych węzłów (aż do wystąpienia następnej gwiazdki). Jeżeli węzeł ma być dodany później (np. za pomocą polecenia konsoli add), a nie podczas uruchomienia środowiska PVM, jego nazwę w pliku konfiguracyjnym poprzedza się znakiem &. Oczywiście, w linijce można pominąć podanie opcji i może ona się ona składać tylko z nazwy maszyny.

Tabela 1 Opcje używane przy dodawaniu nowych węzłów do maszyny wirtualnej PVM

sp	Względna szybkość maszyny. Domyślna wartość: 1000. Uwaga! Praktyka pokazuje, że w ostatnich wersjach PVM wpływ tej opcji jest niezauważalny.
ep	Katalog, w którym znajdują się programy PVM użytkownika. Domyślnie $HOME/pvm3/bin/$PVM_ARCH
dx	Lokalizacja demona PVM na zdalnej maszynie. Domyślnie $PVM_DPATH albo $PVM_ROOT/lib/pvmd
lo	Nazwa konta użytkownika, które ma używać demon PVM. Domyślnie taka sama jak użytkownik na maszynie, z której startuje się środowisko (bądź wykonuje polecenie add)
so	Dodatkowe opcje. Jedyna w praktyce używana opcja pw oznacza wymuszenie podawania hasła.
wd	Katalog roboczy dla uruchamianych programów w środowisku PVM. Domyślna wartość $HOME

Skorzystanie z tego pliku konfiguracyjnego polega na uruchomieniu środowiska PVM (uruchomieniu, nie wznowieniu pracy konsoli!).

```
user@linuxhost:~> pvm machines.conf
```

Przygotowanie programu

Obecnie przygotujemy bardzo prosty program, tak aby zaznajomić się z środowiskiem PVM i sposobem jego uruchamiania.

Program hello.c

```
1. #include "pvm3.h"
2.
3. int main()
4. {
5.    printf("Hello world\n");
6.
7.    pvm_exit();
8. }
```

Dyrektywa preprocesora `include` w linijce 1 załącza plik nagłówkowy z deklaracjami zapowiadającymi funkcji oraz stałych biblioteki PVM. Program wypisuje uświęcony tradycją napis `Hello world` na ekranie, po czym korzystając z funkcji o nazwie `pvm_exit` opuszcza środowisko PVM, po czym się kończy. Dokładniejsze informacje na temat tej funkcji (a także każdej innej) można uzyskać wydając polecenie `man pvm_exit` – o ile został zainstalowany pakiet `pvm-devel`.

Poznane funkcje biblioteki PVM

`int info = pvm_exit(void)`

Funkcja ta informuje lokalny demon *pvmd*, że zadanie opuszcza maszynę wirtualną. Funkcja ta nie kończy wykonywania procesu, który dalej staje się zwykłym procesem systemu UNIX.

Uruchomienie programu

Należy skompilować program za pomocą polecenia:

```
gcc hello.c –o hello –lpvm3
```

Można go uruchomić oczywiście w zwykły sposób za pomocą wydania polecenia `./hello`, ale my skopiujemy go do odpowiedniego katalogu i uruchomimy za pomocą konsoli PVM.

```
cp hello $PVM_HOME
```

Nic nie stoi na przeszkodzie, by uruchomić konsolę PVM i uruchomić nasz pierwszy program PVM. Dokonamy tego za pomocą polecenia konsoli `spawn`. Przykładowy wynik uruchomienia może wyglądać tak:

```
szopen@sirius:~/pvm3/src> pvm
pvm> spawn hello
1 successful
t40002
```

Jak widać program uruchomił się poprawnie i zakończył – jednakże wypisywany napis nie pojawił się na konsoli PVM. Jest to zachowanie prawidłowe, a wyjście programu można obejrzeć w pliku który zazwyczaj posiada nazwę `/tmp/pvml.<numer użytkownika>`. Ponieważ jednak naszym celem było napisanie programu, który coś wypisuje na ekranie, uruchomimy go ponownie, tym razem dodając opcję -> dzięki której powiadamiamy konsolę, że chcemy, by przechwytywała wyjście uruchamianego procesu (oraz jego potomków) i wypisywała je na ekranie.

```
pvm> spawn -> hello
[1]
1 successful
t40003
pvm> [1:t40003] Hello world
[1:t40003] EOF
[1] finished
```

Jak widać, każda linijka poprzedzana jest identyfikatorem zadania PVM w nawiasach kwadratowych. Słowo `EOF` oznacza zakończenie danego procesu *slave*. Zakończenie procesu głównego sygnalizowane jest za pomocą wypisania linijki `finished`.

Program Master-Slave

Kolejnym zadaniem będzie utworzenie prostej aplikacji master-slave. Będzie ona składała się z dwóch części. *Master* będzie uruchamiał programy typu *slave*, które będą wypisywały sakramentalne `hello world!` a następnie kończyły pracę.

Kod *mastera* znajdzie się w pliku o nazwie, jak łatwo odgadnąć, `master.c`.

`Program master.c`

```
1. #include "pvm3.h"
2. #include <stdlib.h>

3. main()
4. {
5.         int mytid;
6.         int tids[4];/*tablica z
                      identyfikatorami procesów slave*/
7.         int nproc;
8.         nproc=pvm_spawn("slave", NULL, PvmTaskDefault, \
                                                  "", 4, tids);
9.         pvm_exit();
10. }
```

Pierwsze dwie linijki to załączenie plików nagłówkowych. Plik `pvm3.h` będzie dołączany do wszystkich programów PVM, natomiast `stdlib.h` załączamy przede wszystkim z uwagi na użycie stałej `NULL`.

Po deklaracjach zmiennych następuje utworzenie 4 procesów *slave* za pomocą funkcji `pvm_spawn` biblioteki PVM. Środowisko same wybiera węzły, na których spróbuje uruchomić programy. Liczba faktycznie uruchomionych procesów zwracana jest przez funkcję i przypisana do zmiennej `nproc`, a ich identyfikatory zostają zapisane do tablicy `tids`.

Programy *slave* w linijce 4 wypisują łańcuch `Hello World!`. W tym momencie należy sobie zadać pytanie, *gdzie* właściwie ma być ten tekst wypisywany, w końcu programy te mogą być odpalane zdalnie na odległych maszynach. Odpowiedź brzmi: wszystko, co jest wypisywane na ekranie jest przechwytywane przez środowisko PVM. Jeżeli tak wskażemy (np. za pomocą opcji -> dla polecenia konsoli `spawn`) wyjście programu wysyłane jest do procesów nadrzędnych (w ostateczności, konsoli). Oznacza to równocześnie, że jeżeli któryś z programów będzie posiadał błędy (na przykład, proces nadrzędny *master* zostanie zablokowany, albo któryś proces ulegnie awarii), to najprawdopodobniej nie zostaną wypisane na ekranie *żadne* komunikaty, co znacząco utrudnia diagnostykę programów PVM.

Każdy program PVM przed zakończeniem powinien wywołać funkcję `pvm_exit`. Funkcja ta nie kończy działania programu, a jedynie powoduje opuszczenie środowiska PVM. W przypadku braku tej

funkcji, mogą na przykład nie być wyświetlane napisy wypisywane na standardowym wyjściu (za pomocą funkcji `printf`).

Program *slave* wygląda wyjątkowo prosto. Jego zawartość powinna się znaleźć w pliku o nazwie `slave.c`.

`Program slave.c`

```
1. #include "pvm3.h"

2. int main()
3. {
4.         printf("Hello world!\n");
5.         pvm_exit();
6. }
```

Poznane funkcje biblioteki PVM

`int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int tids)`

Funkcja `pvm_spawn` uruchamia `ntask` kopii programu o nazwie `task` i rejestruje je w maszynie wirtualnej. `argv` jest wskaźnikiem do tablicy zawierającej argumenty wywołania programu (może on posiadać wartość NULL jeżeli żadne argumenty nie mają być przekazywane do programu). Kombinacja argumentów `flag` oraz `where` pozwala uruchomić zadania na konkretnym węźle, na konkretnej architekturze, lub pozostawia wybór miejsca systemowi w celu równoważenia obciążenia maszyny wirtualnej. Jeżeli argument `flag` posiada wartość `PvmTaskDefault`, to system sam dokonuje wyboru a argument `where` jest ignorowany. Identyfikatory utworzonych procesów wpisywane są do tablicy `tids`. Funkcja zwraca liczbę faktycznie utworzonych procesów.

Kompilacja oraz uruchomienie programu

Należy skompilować program za pomocą polecenia:

```
gcc master.c –o master –lpvm3
gcc slave.c –o slave –lpvm3
```

Następnie należy oba programy skopiować do odpowiedniego katalogu.

```
cp master slave $PVM_HOME
```

Kolejnym krokiem jest uruchomienie konsoli PVM i wydanie polecenia `spawn`. Przykładowy efekt może wyglądać następująco:

```
pvm> spawn master
1 successful
t40005
pvm>
```

Oczywiście, chcielibyśmy widzieć tekst wypisywany w wyniku działania programów typu *slave*. Należy więc dodać opcję -> do komendy `spawn`. Przykładowy rezultat wydania odpowiedniego polecenia może być podobny do przedstawionego poniżej.

```
user@linuxhost:~/pvm3/bin> pvm
pvmd already running.
pvm> spawn -> mst
[1]
1 successful
t140001
pvm> [1:t40004] Hello world!
[1:t140002] Hello world!
[1:t40004] EOF
[1:t140002] EOF
[1:t140001] EOF
[1:tc0002] Hello world!
[1:tc0002] EOF
[1:t100002] Hello world!
[1:t100002] EOF
[1] finished
pvm>
```

Dynamiczne dodawane węzłów

Ostatni program, który napiszesz dzisiaj, będzie służył do dynamicznej dodawania i usuwania węzłów oraz sprawdzania konfiguracji maszyny wirtualnej. Dzięki poznanym funkcjom dowiesz się, w jaki sposób programy mogą dostosowywać maszynę wirtualną do swoich potrzeb.

Do dodawania węzłów służy funkcja `pvm_addhosts`, której argumenty to tablica łańcuchów zawierająca nazwy dodawanych maszyn, rozmiar tablicy oraz tablica, która zostanie wypełniona odpowiednimi identyfikatorami (lub kodami błędów, jeżeli maszyny zdalnej nie da się dodać do maszyny wirtualnej PVM).

Do usuwania węzłów służy funkcja `pvm_delhosts`, której argumenty posiadają znaczenie analogiczne jak w przypadku omówionej funkcji `pvm_addhosts`. Wreszcie, bieżącą konfigurację maszyny wirtualnej można sprawdzić za pomocą funkcji `pvm_config`.

Kod programu przedstawiony jest poniżej.
`Program config.c`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <pvm3.h>

4. int main(int argc, char **argv)
5. {
6.      int info[2];
7.      static char *hosts[]={"lab-143-1", "lab-143-2"};
8.      struct pvmhostinfo * hostp;
9.      int i, nhost, narch;

10.     pvm_addhosts( hosts, 2, info );
11.     pvm_config( &nhost, &narch, &hostp );
12.     printf("Liczba wezlow: %d\n", nhost );
13.     printf("Liczba architektur: %d\n", narch );
14.     for (i=0;i<nhost;i++)
15.     {
16.           printf("Wezel %s ma architekture %s i predkosc \
                %d\n", hostp[i].hi_name, hostp[i].hi_arch, \
                     hostp[i].hi_speed);
17.     }
18.     pvm_delhosts( hosts, 1, info );
19.     pvm_exit();
20.  }
```

Potrzebna jest deklaracja tablicy zawierającej nazwy dodawanych węzłów. Znajduje się ona w linijce 7-mej. Linijka 8 zawiera deklarację zmiennej `hostp`, która będzie zawierać adres struktury z informacjami o maszynie wirtualnej PVM. Dodanie dwóch węzłów (o nazwach „`lab-143-1`" oraz „`lab-143-2`") następuje w linijce 10. W linijce 11 pobierane są informacje na temat konfiguracji maszyny wirtualnej: liczba maszyn (zmienna `nhost`, zarazem liczba wpisów w tablicy `hostp`), liczba różnych architektur (zmienna `narch`), oraz szczegółowe dane na temat poszczególnych węzłów (zmienna `hostp`, której wpisy to struktury zawierające nazwa węzła `hi_name`, nazwa architektury `hi_arch` oraz relatywna prędkość węzła `hi_speed`). Informacje te są wypisywane w linijkach 12-17. Wreszcie, jedna maszyna (o nazwie „`lab-143-1`") jest usuwana. Na końcu proces opuszcza środowisko PVM za pomocą funkcji `pvm_exit()` (linijka 19) i kończy swoje działanie.

Należy skompilować program za pomocą polecenia:

```
gcc config.c –o config –lpvm3
```

Następnie należy program skopiować do odpowiedniego katalogu.

```
cp config $PVM_HOME
```

Załóżmy, że maszyna wirtualna początkowo składa się tylko z jednego węzła, `linuxhost`. Przykładowy efekt uruchomienia za pomocą komendy konsoli PVM spawn pokazany jest poniżej:

```
pvm> spawn -> config
[1]
1 successful
t40002
pvm>
Console: 2 new hosts added
                    HOST     DTID     ARCH   SPEED
                lab-143-1    80000  LINUX64    1000
                lab-143-2    c0000  LINUX64    1000
[1:t40002] Liczba wezlow: 3
[1:t40002] Liczba architektur: 2
[1:t40002] Wezel linuxhost ma architekture LINUX i predkosc 1000
[1:t40002] Wezel lab-143-1 ma architekture LINUX64 i predkosc
1000
[1:t40002] Wezel lab-143-2 ma architekture LINUX64 i predkosc
1000
[1:t40002] EOF
[1] finished
```

Po zakończeniu programu można sprawdzić konfigurację maszyny wirtualnej:

```
pvm> conf
2 hosts, 2 data formats
                    HOST     DTID     ARCH   SPEED       DSIG
                  sirius    40000    LINUX    1000 0x00408841
                lab-143-2    c0000  LINUX64    1000 0x00408c41
```

Jak widać dynamicznie zostały dodane dwa nowe węzły, a następnie jeden z nich został usunięty, co jest odzwierciedlone w konfiguracji pokazanej za pomocą polecenia konsoli PVM `conf`.

Poznane funkcje biblioteki PVM

```
int info = pvm_addhosts(char **hosts, int nhost, int *infos)
int info = pvm_delhosts(char **hosts, int nhost, int *infos)
```

Funkcje te służą do dodawania i usuwania węzłów maszyny wirtualnej. Wartością zwracaną przez funkcję jest ilość węzłów dla których operacja powiodła się. Argument `infos` jest tablicą zawierającą dla każdego węzła status operacji, co w przypadku niepowodzenia umożliwia użytkownikowi określenie przyczyny. Funkcje te stosuje się do uruchamiania maszyny wirtualnej, lecz równie często służą do zwiększenia elastyczności i tolerancji uszkodzeń w przypadku dużych aplikacji. Funkcje te pozwalają aplikacji dostosowywać maszynę wirtualną do swojego zapotrzebowania na moc obliczeniową poprzez dodawanie kolejnych węzłów i usuwanie już niepotrzebnych.

```
int info = pvm_config(int *nhost, int *narch, struct
pvmhostinfo **hostp)
```

Funkcja `pvm_config` zwraca informację o konfiguracji maszyny wirtualnej włączając w to ilość węzłów, oraz różnych architektur. Argument `hostp` jest wskaźnikiem na tablicę struktur `pvmhostinfo`. Każda z tych struktur zawiera *TID* demona, nazwę węzła, rodzaj architektury i relatywną prędkość maszyny, na której działa demon.

Podsumowanie

Podczas tego ćwiczenia udało ci się skonfigurować maszynę wirtualną składającą się z kilku węzłów oraz uruchomić pierwsze, bardzo proste aplikacje napisane z wykorzystaniem bibliotek PVM.

Co powinieneś wiedzieć:

- Jak skonfigurować maszynę wirtualną PVM oraz jak dodawać nowe węzły (polecenie `add` konsoli PVM)
- Jak uruchamiać programy PVM (polecenie `spawn` konsoli PVM)
- Jakie funkcje biblioteczne służą do uruchamiania nowych procesów PVM (`pvm_spawn`) i wychodzenia procesów z środowiska PVM (`pvm_exit`). Powinieneś koniecznie zapamiętać, że każdy program powinien przed zakończeniem wyjść z środowiska PVM.
- W jaki sposób programowo dodawać (`pvm_addhosts`) i usuwać (`pvm_delhosts`) węzły do/z maszyny wirtualnej, oraz jak sprawdzać jej bieżącą konfigurację (`pvm_config`)