Mechanizmy komunikacji grupowej w bibliotece PVM

Zakres ćwiczenia

Celem ćwiczenia jest demonstracja funkcji komunikacji grupowej dostępnych w bibliotece PVM, a także zrozumienie konieczności mechanizmów synchronizacyjnych w systemach rozproszonych, jak również pokazanie dostępnej podstawowej operacji synchronizującej, bariery.

Komunikacja grupowa

W przypadku, gdy procesy pracują wspólnie nad jednym zadaniem, przydatna może być abstrakcyjne pojęcie *grupy* procesów, wspierającej pewien zestaw prostych operacji. Takie operacje mogą obejmować podstawowe operację rozgłaszania (wysyłania wiadomości do wszystkich członków grupy), operacje synchronizacji (zapewniania, że wszyscy członkowie grupy osiągnęli już pewien moment przetwarzania) i tak dalej. Grupy takie mogą być stałe i niezmienne (statyczne), lub też dynamiczne, których skład jest zmienny w czasie. Podczas projektowania mechanizmów komunikacji grupowej w grupach dynamicznych pojawia się cały szereg zagadnień, takich jak problem widoku składu grupy i tak dalej.

W PVM grupy są dynamiczne, co znaczy, że dowolne zadanie może w każdej chwili przyłączyć się do grupy (`pvm_joingroup`), bądź ją opuścić (`pvm_lvgroup`) bez konieczności powiadamiania innych. Dowolne zadanie może wysłać komunikat rozgłoszeniowy do grupy nawet jeśli nie jest w danej chwili jej członkiem. Wyjątkiem są funkcje `pvm_lvgroup`, `pvm_barrier` (omówienie funkcji dalej) i `pvm_reduce` (funkcja ta zostanie omówiona podczas przyszłych ćwiczeń), które ze względu na swoją specyfikę wymagają, aby wywołujące je zadanie było członkiem grupy.

Zarządzaniem grupami w środowisku PVM zajmuje się wydzielony demon, nazywany serwerem grup (`pvmgs`). Jest on automatycznie uruchamiany, jeżeli następuje taka potrzeba. Grupy w PVM są więc zarządzane w sposób scentralizowany, co powoduje, że ich rozmiar nie powinien być zbyt duży – gdyż przy rosnącym rozmiarze grup serwer grup staje się wąskim gardłem przetwarzania. Ogólną filozofią PVM-a jest jednak prostota i przezroczystość rozwiązań kosztem nawet efektywności, więc takie rozwiązanie dobrze się w nią wpasowuje.

Oprócz funkcji poznanych w trakcie obecnych zajęć, PVM udostępnia także szereg funkcji bardziej zaawansowanych, które zostaną omówione w przyszłości. Funkcje te są przydatne do bardziej zwięzłego zapisu niektórych podstawowych operacji często używanych w tworzeniu aplikacji rozproszonych.

Rozgłaszanie w środowisku PVM

Utworzymy obecnie prosty program, który będzie wykorzystywał funkcje dostarczane przez bibliotekę PVM do utworzenia grupy i rozesłania do grupy komunikatów. Jak zwykle będzie się on składał z dwóch części: *mastera* i procesów typu *slave.* Wszystkie te procesy dołączą do utworzonej grupy. Proces *master* roześle do wszystkich członków grupy wiadomość a następnie poczeka na wiadomości potwierdzające ich odbiór.

Program ten będzie posiadał pewien mały błąd, wprowadzony celowo w celach dydaktycznych. Pokażemy na czym polega ten błąd oraz w jaki sposób go naprawić – co dokonasz już samodzielnie. Da to nam również okazję do poznania kilku nowych poleceń konsoli PVM, służących do operacji na zadaniach PVM.

Pierwszym plikiem będzie plik `def.h`, w którym znajdzie się jedna dodatkowa definicja: `GROUPNAME`, czyli nazwa grupy (linijka 6). Pozostała część pozostanie identyczna jak poprzednio, można więc wykorzystać poprzednio utworzony plik dopisując do niego jedną linijkę.

`plik def.h`

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <pvm3.h>
4. #define SLAVENAME "slave"
5. #define SLAVENUM   4
6. #define GROUPNAME "grupa"
7. #define NAMESIZE   64
8. #define MSG_MSTR 1
9. #define MSG_SLV  2
```

Pierwsza część programu *mastera* nie będzie się różnić zbytnio od poprzednich. Widzimy więc załączenie plików nagłówkowych, definicje zmiennych i utworzenie `SLAVENUM` procesów typu *slave*.

`program master.c`

```
1. #include "def.h"
2. #include <stdio.h>
3.
4. int main()
5. {
6.      int tid = pvm_mytid();
7.      int tids[SLAVENUM];
8.      int i, res=0;
9.
10.     pvm_spawn("slave",0,0,".",SLAVENUM,tids);
```

Następnie następuje do dołączenie do grupy za pomocą funkcji `pvm_joingroup` (linijka 9). Jedyny argument tej funkcji to łańcuch tekstowy, nazwa grupy do której proces chce się dołączyć, tutaj zdefiniowanej za pomocą `GROUPNAME` (która w pliku `def.h` została określona jako po prostu „grupa”). Pierwszy proces próbujący się dołączyć do grupy powoduje równocześnie jej utworzenie. Funkcja ta zwraca numer instancji procesu w grupie – liczbę całkowitą jednoznacznie identyfikującą ten proces w obrębie danej grupy. Oznacza to więc, że para <nazwa grupy, numer instancji> jest unikalny w obrębie danej maszyny wirtualnej. Założyciel grupy otrzyma numer instancji 0. Nie jest jednak zagwarantowane, że proces o numerze instancji równym zero to założyciel grupy - każdy proces dołączający do grupy otrzymuje najmniejszy dostępny numer instancji, co oznacza, że jeżeli założyciel opuści grupę za pomocą funkcji `pvm_lvgroup`, to kolejny proces otrzyma jego numer.

Kolejne trzy linijki (10-12) to rozesłanie komunikatu do całej nowo utworzonej grupy. Odbywa się to w podobny sposób jak wysłanie komunikatu do pojedynczego procesu. Najpierw tworzony jest bufor wiadomości za pomocą funkcji `pvm_initsend`, następnie w tym buforze umieszczany jest element typu `int` za pomocą funkcji `pvm_pkint` (tutaj identyfikator *mastera*). Różnica polega na wywołaniu funkcji `pvm_bcast` zamiast `pvm_send`. Argumentem tej funkcji jest nazwa grupy oraz znacznik wiadomości.

```
9.     pvm_joingroup( GROUPNAME );
10.    pvm_initsend( PvmDataRaw );
11.    pvm_pkint( &tid, 1,1);
12.    pvm_bcast( GROUPNAME, MSG_MSTR);
```

Bieżący rozmiar grupy można pobrać za pomocą funkcji `pvm_gsize`, której argumentem jest nazwa grupy.

Następnie proces *master* oczekuje od wszystkich procesów na potwierdzenia (w dowolnej kolejności). W tym celu w pętli (linijki 13-18) wywołuje blokującą funkcji odbioru wiadomości `pvm_recv`, deklarując odbiór od dowolnego procesu i o typie wiadomości `MSG_SLV`. Po nadejściu wiadomości wypakowywana jest z niej liczba typu `int` (którą, jak zobaczymy analizując kod procesu *slave*, jest identyfikator nadawcy), która zostaje wypisana na ekranie.

```
13.     for (i=0;i<SLAVENUM;i++)
14.     {
15.           pvm_recv( -1, MSG_SLV );
16.           pvm_upkint( &res, 1, 1);
17.           printf("Master: %d-ty komunikat. \
                    Nadawca to t%x\n",i,res);
18.     }
19.     printf("Master: Odebralem wiadomosc od\
                                    wszystkich\n");
20.     pvm_lvgroup(GROUPNAME);
21.     pvm_exit();
22.   }
```

Wreszcie, po wypisaniu na standardowe wyjście komunikatu o odebraniu wiadomości od wszystkich procesów, proces opuszcza grupę (co nie jest konieczne, ponieważ zaraz i tak się skończy – chodzi tutaj tylko o demonstrację użycia funkcji `pvm_lvgroup`) i następuje wyjście z środowiska PVM oraz zakończenie programu.

Procesy *slave* będą wykonywały program, którego kod zapiszemy w pliku `slave.c`. Zadaniem każdego programu *slave* będzie odebranie wiadomości od *mastera* i następnie odesłanie mu w odpowiedzi, jako potwierdzenia, własnego identyfikatora.

`Program slave.c`

```
23.   #include "def.h"
24.   #include <stdio.h>
25.
26.   int main()
27.   {
28.     int ptid = 0;
29.     int tid = pvm_mytid();
30.     printf("Slave: Czekam na wiadomosci\n");
31.     pvm_joingroup( GROUPNAME );
32.     pvm_recv( pvm_parent(), MSG_MSTR );
33.     pvm_upkint( &ptid, 1, 1);
34.     pvm_initsend( PvmDataRaw );
35.     pvm_pkint( &tid, 1, 1);
36.     pvm_send( pvm_parent(), MSG_SLV);
37.     pvm_exit();
38.   }
```

Pierwszy fragment wymagający wyjaśnienia zaczyna się w linijce 29-tej. Proces dołącza do grupy za pomocą funkcji `pvm_joingroup` (być może ją tworząc, jeżeli był przypadkiem pierwszym wywołującym tą funkcję). Następnie blokuje się w oczekiwaniu na wiadomość od procesu macierzystego o typie `MSG_MSTR`. Po otrzymaniu tej wiadomości tworzy bufor komunikacyjny (funkcja `pvm_initsend`, linijka 32), umieszcza w nim swój identyfikator (funkcja `pvm_pkint`, linijka 33) i wysyła go do procesu macierzystego z znacznikiem wiadomości `MSG_SLV` (funkcja `pvm_send`, linijka 34). Wreszcie opuszcza środowisko PVM za pomocą funkcji `pvm_exit` (linijka 35) i kończy swoje działanie.

Program ten, co może niełatwo zauważyć, jest błędny. Przed przejściem do dalszej części ćwiczenia (kompilacji i uruchomienia programu) spróbuj zastanowić się, co jest tego przyczyną. Tak

skonstruowana aplikacja nigdy się nie zakończy. Czy potrafisz powiedzieć, w którym miejscu nastąpi zablokowanie aplikacji i dlaczego?

Teraz skompilujemy oba programy. Pojawia się tutaj niewielka różnica w stosunku do poprzednich ćwiczeń. Ponieważ wykorzystują one funkcje komunikacji grupowej, powinieneś w czasie kompilacji dodać dodatkową opcję `-lgpvm3`, powiadamiającą program konsolidujący o konieczności skorzystania z biblioteki współdzielonej zawierającej ciała funkcji służących do różnorakich operacjach na grupach PVM:

```
gcc master.c -o master -lpvm3 -lgpvm3
gcc slave.c -o slave -lpvm3 -lgpvm3
```

Następnie utworzone pliki wykonywalne kopiujemy do odpowiedniego katalogu

```
cp master slave $PVM_HOME
```

Aplikację można teraz uruchomić za pomocą polecenia konsoli PVM `spawn -> master`. Wynik uruchomienia (dla czterech procesów typu *slave*) może wyglądać na przykład tak:

```
pvm> spawn -> master
[1]
1 successful
t80016
pvm> [1:t4001e] Slave: Czekam na wiadomosci
[1:t4001e] EOF
[1:t80017] Slave: Czekam na wiadomosci
[1:t80017] EOF
pvm>
```

Jak widać, nie pojawił się napis `[1] finished`, który oznacza zakończenie z sukcesem aplikacji. Widzimy w tym przykładzie także tylko dwa napisy `EOF`, co oznacza, że tylko dwa procesy *slave* zakończyły swoje działanie. Aplikacja więc się nie skończyła. Dlaczego?

Spróbujmy zobaczyć, jakie procesy znajdują się obecnie w środowisku PVM. Dowiemy się tego za pomocą komendy konsoli `ps`. Samo `ps` powoduje wypisanie zadań PVM na bieżącym węźle, natomiast `ps -a` wypisze zadania na całej maszynie wirtualnej PVM.

```
pvm> ps -a
                HOST       TID   FLAG 0x COMMAND
             lab-143-3    c0015     6/c,f slave
             lab-143-3    c0016     6/c,f slave
             lab-143-3    c0017     6/c,f pvmgs
             lab-143-2    80016     6/c,f master
```

Jak widać, w systemie wciąż istnieją trzy (w tym wypadku) procesy: dwa typu *slave* i jeden typu *master*. Analiza kodu programu powinna doprowadzić ciebie do wniosku, że mogą się one zablokować tylko w jednym miejscu: w czasie wywołania blokującej funkcji `pvm_recv` (linijka 15 w `master.c` oraz linijka 30 w `slave.c`). Procesy najwidoczniej oczekują na pojawienie się komunikatów, które nigdy nie nadchodzą. Pytanie brzmi: dlaczego? Nie należy od razu wysnuwać wniosków o tym, że środowisko PVM „zgubiło” jakiś komunikat. Jest to praktycznie niemożliwe. Analiza kodu na pewno doprowadzi nas do wniosku, że komunikaty o oczekiwanym typie i identyfikatorze *powinny* być wysyłane. Najwidoczniej jednak nie są.

Rozwiązanie tej zagadki jest niezwykle proste. Mianowicie, kiedy proces *master* rozsyła wiadomość do wszystkich członków grupy, jest możliwe, że jeszcze nie wszystkie procesy *slave* do niej zdążyły dołączyć. W takim wypadku może zajść sytuacja, w której *master* rozsyła wiadomość do członków grupy obejmującej dopiero część procesów *slave*, po czym zaczyna oczekiwać na wiadomości od wszystkich procesów. Tymczasem część procesów *slave* dopiero wtedy dołącza do grupy (więc wiadomość rozesłana wcześniej przez *mastera* nigdy do nich nie dotrze) i blokuje się oczekując na przybycie wiadomość (a więc nigdy nie wyśle do *mastera* oczekiwanego przez niego potwierdzenia).

Oczywiście, musimy wziąć pod uwagę, że środowisko jest rozproszone, więc przedstawiona wyżej sytuacja nie musi się zdarzyć. Istnieje możliwość, że przypadkowo wszystkie procesy *slave* zdążą dołączyć do grupy zanim *master* zacznie rozsyłać wiadomości, a więc, że program zakończy się sukcesem (możliwość ta jest nawet całkiem duża, jeżeli zapomniałeś dodać jakieś dodatkowe węzły do maszyny wirtualnej i uruchamiasz przypadkiem programy na tylko jednym komputerze). Jednakże dobrzy programiści nie mogą zakładać, że ich programy *być może* będą działać. Dobrzy programiści powinni pisać programy tak, by działały *zawsze* (a przynajmniej, zawsze oprócz naprawdę patologicznych sytuacji).

Zanim przystąpimy do napisania poprawnej wersji programu, należy przerwać zadania PVM. Można to uczynić albo za pomocą komendy konsoli `kill`, albo użyć komendy `reset`. Komenda `kill` służy do kończenia pojedynczych zadań działających w środowisku PVM. Należy tutaj podkreślić, że taka komenda *jest* konieczna – co często nie jest oczywiste dla niektórych studentów, którzy zapominają, że zadania PVM są rozproszone na różnych, odległych komputerach, więc zabicie ręcznie ich wymagałoby wyszukiwania odpowiednich procesów i ich zabijanie za pomocą polecenia `kill` systemu operacyjnego każdego z węzłów. Z kolei komenda `reset` służy, jak sama nazwa wskazuje, do resetowania maszyny wirtualnej PVM. Zatrzymuje ona wszystkie aktualnie działające zadania.

```
pvm> reset
pvm> [1:t80016] EOF
[1:tc0015] EOF
[1:tc0016] EOF
[1] finished
```

Dla bezpieczeństwa, przed uruchomieniem wszystkich programów powinieneś resetować maszynę wirtualną PVM. Zapewni to, że na wpływ wykonywanego programu nie wpływają procesy, które pozostały na maszynie po ewentualnym zablokowaniu w czasie jednej z możliwych błędnych realizacji wykonywanych zadań.

Zadanie do samodzielnego wykonania

Jak widzimy, podstawowym problemem w poprzedniej realizacji wyznaczonego przez nas zadania był fakt braku synchronizacji między procesami. Proces *master* za wcześnie rozsyłał wiadomości, nie czekając do chwili w której wszystkie procesy *slave* znalazły się w grupie. Należy więc wstrzymać się z rozsyłaniem dopóki *master* nie będzie miał pewności, że pozostałe procesy są gotowe na odbiór wiadomości. Można oczywiście zrobić to na przykład za pomocą wprowadzenia kolejnej pętli w pliku `master.c`, w której proces *master* czekałby na wiadomości od wszystkich pozostałych, mówiące mniej więcej: „Jestem gotowy! Zacznij rozsyłanie!”. Nie należy jednak wynajdywać koła w sytuacji, w której PVM dostarcza już własnych gotowych, prostych mechanizmów.

Rozwiązaniem jest użycie do synchronizacji procesów funkcji `pvm_barrier`. Ta dwuargumentowa funkcja służy do upewnienia się, że wszystkie procesy grupy osiągnęły już pewien moment przetwarzania. Jej wywołanie jest blokujące. Pierwszy argument to nazwa grupy, a drugi to liczba procesów, które powinny również wywołać barierę w tej grupie aby można było odblokować proces. Tak więc, jeżeli proces wywoła funkcję `pvm_barrier("pracusie", 4)`, to będzie on wstrzymany dopóki łącznie cztery procesy z grupy „pracusie” (włącznie z nim samym, jeżeli też należy do grupy!) nie wywołają tej funkcji. Dodatkową zaletą jej użycia jest jasna semantyka operacji. Programista analizując kod będzie od razu wiedział, co robi ta linijka – co nie musiałoby być prawdą, gdybyśmy chcieli osiągnąć to samo za pomocą blokującej funkcji `pvm_recv`.

Rozwiązanie zadania znajduje się w materiałach kursu. Nie zaglądaj do niego, dopóki nie ukończysz ćwiczenia samodzielnie. Wynik uruchomionego programu (dla czterech procesów typu *slave*) może wyglądać na przykład tak:

```
pvm> spawn -> master
[1]
1 successful
t40016
pvm> [1:t40017] Slave: Czekam na wiadomosci
[1:t40017] Slave: Nazwa grupy: grupa
[1:t40017] Slave: Jestem za bariera
[1:t40017] EOF
[1:tc000e] Slave: Czekam na wiadomosci
[1:tc000e] Slave: Nazwa grupy: grupa
[1:tc000e] Slave: Jestem za bariera
[1:t80011] Slave: Czekam na wiadomosci
[1:t80011] Slave: Nazwa grupy: grupa
[1:t80011] Slave: Jestem za bariera
[1:tc000e] EOF
[1:t40016] Master: Czekam na barierze
[1:t40016] Master: Jestem za Bariera
[1:t40016] Master: 0-ty komunikat. Nadawca to t40017
[1:t40016] Master: 1-ty komunikat. Nadawca to tc000e
[1:t40016] Master: 2-ty komunikat. Nadawca to t80011
[1:t40016] Master: 3-ty komunikat. Nadawca to t80012
[1:t40016] Master: Odebralem wiadomosc od wszystkich
[1:t40016] EOF
[1:t80012] Slave: Czekam na wiadomosci
[1:t80012] Slave: Nazwa grupy: grupa
[1:t80012] Slave: Jestem za bariera
[1:t80011] EOF
[1:t80012] EOF
[1] finished
```

Poznane funkcje biblioteki PVM

`int inum = pvm_joingroup(char *group)`

`int info = pvm_lvgroup(char *group)`

Funkcje te pozwalają zadaniu przyłączyć się do grupy, lub ją opuścić. Pierwsze wywołanie funkcji `pvm_joingroup` dla grupy powoduje utworzenie danej grupy i dodanie do niej zadania. Funkcja `pvm_joingroup` zwraca również pozycję procesu w grupie. W PVM proces może należeć do wielu grup. W przypadku opuszczenia grupy i ponownego się do niej przyłączenia zadanie może mieć nową pozycję.

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

Po wywołaniu funkcji `pvm_barrier` proces wstrzymywany jest do chwili, aż `count` procesów należących do grupy wywoła synchronicznie funkcję `pvm_barrier`. W większości przypadków `count` równe jest ilości procesów w grupie.

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

Funkcja `pvm_bcast` wysyła komunikat zaopatrzony w etykietę `tag` do wszystkich procesów należących do grupy z wyjątkiem siebie samego. Ponieważ grupy są dynamiczne przyłączenie nowego procesu do grupy podczas rozgłaszania może spowodować, że komunikat nie zostanie przez ten proces odebrany. Podobnie jeśli proces opuści grupę po rozpoczęciu operacji rozgłaszania i tak otrzyma komunikat.

Stworzenie topologii pierścienia

Kolejnym zadaniem do wykonania dzisiaj będzie modyfikacja programu implementującego topologię pierścienia wykonanego na jednych z poprzednich ćwiczeń. Tym razem wykorzystamy do tego mechanizmy grup udostępniane przez środowisko PVM, co znacznie ułatwi nam zadanie.

Wykorzystamy trzy dodatkowe funkcje. Pierwsza z nich to `pvm_gettid`, która podaje identyfikator zadania PVM na podstawie podanej nazwy grupy i numeru instancji w grupie. Druga to `pvm_getinst`, która z kolei podaje numer instancji procesu o zadanym identyfikatorze w grupie podanej jako parametr. Wreszcie ostatnia to `pvm_gsize`, która podaje rozmiar grupy określonej przez parametr.

`Program master.c`

```
1. #include "def.h"

2.  int main(int argc, char **argv)
3.  {
4.    int tids[SLAVENUM];
5.    int res = 0;

6.    pvm_joingroup(GROUPNAME);
7.    pvm_spawn("slave",0,0,".",SLAVENUM,tids);
8.    pvm_barrier(GROUPNAME, SLAVENUM+1);
9.    pvm_initsend( PvmDataRaw );
10.   pvm_pkint( &res, 1,1);
11.   pvm_send( pvm_gettid( GROUPNAME, 1), MSG_SLV);
12.   pvm_recv( -1, MSG_SLV );
13.   pvm_upkint( &res, 1, 1);
14.   printf("Master: Token przeszedl pierscien: t%d\n",res);
15.   pvm_exit();
16.  }
```

W nowej wersji programu procesy jako swojego następnika będą wybierały kolejny proces w grupie (według numeru instancji). Aby zapewnić, by *master* otrzymał numer instancji równy zero, dołączy on do grupy jako pierwszy, przed utworzeniem jakichkolwiek innych procesów (linijka 6).

Następnie *master* czeka, aż wszystkie procesy dołączą do grupy (funkcja `pvm_barrier`, linijka 8). Jego następnikiem w pierścieniu ma być proces o numerze instancji w grupie równym 1. Aby dowiedzieć się, jaki ten proces ma identyfikator, wywołuje funkcję `pvm_gettid`, podając jako argumenty nazwę grupy oraz numer instancji równy 1. Dalsza część kodu powinna ci być już znajoma z poprzedniego ćwiczenia realizującego połączenie procesów w pierścień.

Kod procesów typu *slave* jest bardzo podobny. Procesy te wybierają na podobnej zasadzie swojego następnika w pierścieniu, odbierają od poprzednika wiadomość z licznikiem odwiedzin (właściwie oczekują na tą wiadomość od dowolnego procesu, ale jedynym procesem, który może im ją wysłać, jest poprzednik), inkrementują ją i przesyłają dalej.

Program slave.c

```
1. #include "def.h"

2.  int main()
3.  {
4.    int succ = 0, res, i;
5.    int tid = pvm_mytid();
6.    int tids[SLAVENUM];
7.    int ginst = pvm_joingroup(GROUPNAME);

8.    pvm_barrier( GROUPNAME, SLAVENUM+1);
9.    if (ginst = pvm_gsize(GROUPNAME)-1)
10.           succ = pvm_parent();
11.   else
12.           succ = pvm_gettid( GROUPNAME, ginst+1);
13.   printf("Slave: nastepnik w pierscieniu: t%x\n", succ);

14.   pvm_recv( -1, MSG_SLV );
15.   pvm_upkint( &res, 1, 1);
16.   res++;
17.   pvm_initsend( PvmDataRaw );
18.   pvm_pkint( &res, 1, 1);
19.   pvm_send( succ, MSG_SLV );
20.   pvm_exit();
21.  }
```

Wybór następnika następuje w linijkach 9-12. Proces *slave* sprawdza, czy jest ostatnim procesem w grupie, jeżeli tak, to jego następnikiem jest proces macierzysty, w przeciwnym wypadku wybiera następnika w identyczny sposób jak jego *master*. Dalsza część kodu jest ci znajoma z poprzednich ćwiczeń i dlatego nie ma potrzeby jej tutaj omawiać.

Przedstawiony kod nie jest całkowicie poprawny i może teoretycznie zawieść w bardzo szczególnym przypadku. Spróbuj domyśleć się dlaczego oraz w jaki sposób go poprawić, by program działał zawsze (wskazówka: jak się będzie zmieniać rozmiar grupy?).
Wynik poprawionego programu, po skompilowaniu i uruchomieniu z poziomu konsoli PVM może wyglądać tak:

```
pvm> spawn -> master
[1]
1 successful
t4005c
pvm> [1:t4005d] Slave: nastepnik w pierscieniu: t10003c
[1:t10003c] Slave: nastepnik w pierscieniu: t14003c
[1:t4005d] EOF
[1:t10003c] EOF
[1:t14003c] Slave: nastepnik w pierscieniu: tc003d
[1:t14003c] EOF
[1:t4005c] Master: Token przeszedl pierscien: t4
[1:tc003d] Slave: nastepnik w pierscieniu: t4005c
[1:tc003d] EOF
[1:t4005c] EOF
[1] finished
```

Poznane funkcje biblioteki PVM

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

Funkcja `pvm_gettid` zwraca identyfikator procesu należącego do podanej grupy i znajdującego się na określonej pozycji. Funkcja `pvm_getinst` zwraca pozycje w grupie procesu o identyfikatorze tid. Ostatnia funkcja `pvm_gsize` zwraca liczność dynamicznej grupy.

Podsumowanie

Czytając te słowa powinieneś posiadać intuicję na temat niebezpieczeństw wynikających z braku synchronizacji w aplikacjach rozproszonych oraz rozumieć sposób używania barier w środowisku PVM.

Co powinieneś wiedzieć:

- Co to jest bariera i jaka funkcja ją implementuje w bibliotece PVM (`pvm_barrier`)
- Jakie funkcje służą do tworzenia i dołączania do grupy (`pvm_joingroup`), opuszczania grupy (`pvm_lvgroup`)
- Jaka funkcja biblioteki PVM służy do rozgłaszania (`pvm_bcast`)
- W jaki sposób można pobrać rozmiar grupy (`pvm_gsize`) oraz numer instancji w grupie dowolnego procesu – w szczególności swój własny (`pvm_getinst`), a także identyfikator zadania o danym numerze instancji w danej grupie (`pvm_gettid`)
- W jaki sposób usunąć zablokowane procesy (resetować maszynę wirtualną – komenda konsoli `reset`)