Podstawy komunikacji między procesami PVM

Zakres ćwiczenia

Komunikacja między zadaniami PVM odbywa się przy pomocy wymiany wiadomości. Celem ćwiczenia jest więc poznanie podstawowych operacji tworzenia, wysyłania i odbierania wiadomości.

Komunikacja między zadaniami w środowisku PVM

Wysłanie komunikatu składa się w PVM z trzech kroków:

1. Konieczna jest inicjalizacja bufora za pomocą funkcji `pvm_initsend` (lub za pomocą nie omawianej tutaj funkcji `pvm_mkbuf`).
2. Komunikat musi zostać umieszczony w buforze za pomocą kombinacji funkcji `pvm_pk*`.
3. Kompletny komunikat przesyłany jest do innego zadania poprzez wywołanie funkcji `pvm_send` (lub do wielu zadań za pomocą `pvm_mcast`)

Komunikat odbierany jest przez wywołanie blokującej bądź nieblokującej funkcji odbiorczej i późniejsze rozpakowanie wiadomości z bufora do zmiennych. Odbierane komunikaty mogą być filtrowane na podstawie identyfikatora nadawcy i/lub etykiety.

Dane są przesyłane w postaci niezależnej od architektury, chyba, że użytkownik zażąda inaczej.

Środowisko PVM zapewnia kolejność FIFO wiadomości pomiędzy każdymi dwoma komunikującymi się zadaniami. Dzięki mechanizmowi filtrowania wiadomości kolejność ta może być zmieniana.

Koncepcja programu

Aby zademonstrować mechanizmy komunikacji w środowisku PVM, zmodyfikujemy poprzednio utworzoną parę programów *master-slave*. Program *master* będzie tworzył pewną liczbę programów typu *slave*, a następnie będzie im wysyłał swój identyfikator i kolejne liczby poczynając od 0. Następnie będzie oczekiwał na odebranie wiadomości od programów typu *slave*. Aby przekonać się, że faktycznie programy są uruchamiane na różnych węzłach, każdy *slave* będzie odsyłał nazwę maszyny, na której jest uruchomiony, oraz swój identyfikator.

Przed dalszą pracą, uruchom środowisko pvm i dodaj do niego kilka węzłów. Sprawdź za pomocą polecenia `conf` konfigurację maszyny wirtualnej. Wyjdź z konsoli PVM za pomocą komendy `quit`.

Przygotowanie programu

Nasz program składać się będzie z trzech plików: `def.h`, `master.c`, `slave.c`. Pierwszy z tych plików będzie zawierał definicje wspólne dla programów *master* oraz *slave*. Dwie pierwsze linijki to załączenie plików nagłówkowych `pvm3.h` oraz `stdlib.h`. Następnie są zdefiniowane stałe określające nazwę programu wykonywalnego programu *slave*, liczbę zadań *slave* oraz rozmiar maksymalny przewidywany na nazwę maszyny, a także określimy nazwy symboliczne dla liczb wyznaczających typy wiadomości (pojęcie wyjaśnione w dalszej części ćwiczenia).

Program def.h

```
1. #include "pvm3.h"
2. #include <stdlib.h>
3. #define SLAVENAME "slave"
4. #define SLAVENUM 4
5. #define NAMESIZE 64
6. #define MSG_MST 1
7. #define MSG_SLV 2
```

W programie `master.c` pierwsze linijki są identyczne niemal jak ostatnio. Uruchamiane jest kilka programów typu *slave*. Następnie w linijce 9 *master* pobiera własny identyfikator za pomocą funkcji `pvm_mytid`. Kolejne kilka linijek omówimy dokładniej, gdyż zawierają one przygotowanie i wysłanie komunikatu.

Program master.c

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

2. main()
3. {
4.      int mytid;
5.      int tids[SLAVENUM];/* identyfikatory zadań slave */
6.      char slave_name[NAMESIZE];
7.      int nproc, i, who;
8.      nproc=pvm_spawn(SLAVENAME, NULL, PvmTaskDefault, \
                        "", SLAVENUM, tids);
9.      mytid = pvm_mytid();
10.     printf("Master: Moj identyfikator to %d \n", mytid);
```

Jak pamiętamy, funkcja `pvm_spawn` zwraca liczbę faktycznie utworzonych procesów typu *slave*. Tak więc pętla obejmująca linijki 11-17 wysyła komunikaty do wszystkich utworzonych procesów. Na początku należy przygotować bufor komunikacyjny za pomocą funkcji `pvm_initsend`. Funkcja ta tworzy bufor, o ile on wcześniej nie istniał, lub też opróżnia bufor komunikacyjny, jeżeli był on już wcześniej przygotowany. Flaga `PvmDataDefault` określa, że dane przy wkładaniu do bufora mają być konwertowane do postaci niezależnej od architektury.

Po przygotowaniu bufora, można zacząć umieszczać w nim dane za pomocą funkcji z rodziny `pvm_pk*`. Nazwa każdej z takich funkcji składa się z przedrostka `pvm_pk` oraz nazwy typu: np. `pvm_pkint`, `pvm_pkbyte` czy też `pvm_pkstr`. W linijce 4 zapakowany jest 1 element typu `int` do bufora, tak samo w linijce 5. Argumenty funkcji `pvm_pkint` to kolejno adres, pod którym znajduje się element, liczba element, i tzw. odstęp (co który element ma być pakowany: np. można kazać wybierać tylko parzyste elementy tablic) – w wszystkich naszych programach ostatni parametr zawsze będzie równy 1 (za wyjątkiem specjalnego przykładu ilustrującego jego użycie pod koniec obecnego ćwiczenia).

Wreszcie tak przygotowany komunikat jest wysyłany za pomocą funkcji `pvm_send`. Pierwszy argument tej funkcji to adresat wiadomości – tutaj kolejny element tablicy `tids`, która, jak pamiętamy, zawiera identyfikatory zadań uruchomionych przez wywołanie funkcji `pvm_send` w linijce 8 - a drugi to *znacznik* wiadomości, liczba określająca jej typ. Typ wiadomości ma znaczenie dla programisty i umożliwia następnie filtrowanie nadchodzących wiadomości według ich typów.

Można bufor traktować jako pudełko. Funkcja `pvm_initsend` daje nam takie pudełko, lub wyrzuca jego poprzednią zawartość, a funkcje z rodziny `pvm_pk*` wkładają do tego przygotowanego pudełka kolejne elementy do wysłania.

```
11.     for( i=0 ; i<nproc ; i++ )
12.     {
13.           pvm_initsend(PvmDataDefault);
14.           pvm_pkint(&mytid, 1, 1);
15.           pvm_pkint(&i, 1, 1);
16.           pvm_send(tids[i], MSG_MSTR);
17.     }
```

Kolejną czynnością *mastera*, jak już wspomnieliśmy wcześniej, ma być odebranie komunikatów od zadań potomnych. Dzieje się to w pętli w linijkach 18-23. W linijce 20 zadanie blokuje się tak długo, dopóki nie nadejdzie komunikat od dowolnego nadawcy (-1 jako pierwszy argument funkcji `pvm_recv`) oraz o typie MSG_SLV (drugi argument tej funkcji). Przesłane elementy są wypakowane z wiadomości za pomocą funkcji `pvm_upkint`, która wyciąga jeden element typu `int` i umieszcza go w zmiennej `who`. W linijce 21 następuje wypakowanie nadesłanego łańcucha. W tym wypadku nie podaje się liczby elementów ani odstępu; każdy łańcuch zawsze kończy się zerem, więc to jednoznacznie określa, co ma zawierać wypakowany łańcuch.

Należy tutaj podkreślić, że to od programisty zależy, czy w wiadomości faktycznie znajdują się elementy tego typu (i w takiej kolejności), jak tego oczekują wywoływane funkcje z rodziny `pvm_upk*`. W przypadku używania wielu różnych wiadomości różnych typów należy używać znaczników wiadomości.

Wreszcie program opuszcza środowisko pvm (z pomocą `pvm_exit` w linijce 24), po czym kończy swoje działanie.

```
18.           for( i=0 ; i<nproc ; i++ )
19.           {
20.                 pvm_recv( -1, MSG_SLV );
21.                 pvm_upkint(&who, 1, 1 );
22.                 pvm_upkstr(slave_name );
23.                 printf("Master: proces t%x jest na hoscie \
                              %s\n",who,  slave_name);
24.           }
25.           pvm_exit();
26.   }
```

Przyjrzyjmy się teraz strukturze programu *slave*. Będzie on odbierał wiadomość od *mastera*, a następnie odsyłał mu nazwę maszyny (*hostname*) na której został uruchomiony razem z własnym identyfikatorem.

Pierwsze linijki zasadniczo nie wymagają wyjaśnienia – są to załączenie pliku nagłówkowego `def.h`, deklaracje zmiennych oraz przydział miejsca w sposób dynamiczny dla zmiennej łańcuchowej `str`.

`Program slave.c`

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

2.   int main()
3.   {
4.      int ptid, my2;
5.      int mytid;
6.      char *str = malloc( 64 );
```

Następnie w linijce 7 *slave* blokuje się w oczekiwaniu na nadejście dowolnej wiadomości o znaczniku `MSG_MSTR`. Gdy taka wiadomość nadejdzie, wypakowuje z niej dwie liczby typu `int` i wypisuje je na standardowym wyjściu. W linijce 16 pobiera swój własny identyfikator i przypisuje go do zmiennej `mytid`. Następnie tworzy bufor komunikacyjny, pakuje do niego swój identyfikator, pobiera nazwę maszyny na której się znajduje w linijce 19 i pakuje go do bufora (linijka 21). W funkcji `pvm_pkstr` nie jest potrzebne określenie rozmiaru łańcucha – jest on określany automatycznie, gdyż, jak pamiętamy, łańcuchy w języku C są kończone zerem. Funkcja `gethostbyname` właściwie kończy łańcuch zerem, za wyjątkiem przypadku specjalnego, gdy nie mieści się on w podanej długości znaków – 64 w podanym przykładzie. Z uwagi na ten przypadek specjalny dodana jest linijka 20. Następnie tak przygotowany komunikat wysyłany jest do ojca. Identyfikator ojca znamy, gdyż został on nadesłany w odebranym wcześniej komunikacie. Na końcu proces *slave* wychodzi z środowiska PVM i kończy pracę.

```
7.      int i = pvm_recv( -1, MSG_MSTR);
8.      if ( i < 0 )
9.      {
10.           perror("Nieoczekiwany błąd.");
11.           exit(0);
12.     }
13.     pvm_upkint( &ptid, 1, 1);
14.     pvm_upkint( &my2, 1, 1);
15.     printf("Slave: Otrzymałem %x %d \n", tid, my2);
16.     mytid = pvm_mytid();
17.     pvm_initsend( PvmDataDefault);
18.     pvm_pkint( &mytid, 1, 1);
19.     gethostname( str, 64 );
20.     str[64] = 0;
21.     pvm_pkstr( str );
22.     pvm_send(mynumber, MSG_SLV);
23.     pvm_exit();
24.  }
```

Należy teraz skompilować oba programy i umieścić pliki wykonywalne w odpowiednim katalogu:

```
gcc master.c –o master –lpvm3

gcc slave.c –o slave –lpvm3

cp master slave $PVM_HOME
```

Następnie można je uruchomić za pomocą polecenia konsoli PVM `spawn -> master`. Wynik uruchomienia może wyglądać na przykład tak:

```
pvm> spawn -> master
[1]
1 successful
t40020
pvm> [1:t40021] Slave: Otrzymałem 40020 3
[1:t40021] EOF
[1:tc0019] Slave: Otrzymałem 40020 2
[1:tc0019] EOF
[1:t80018] Slave: Otrzymałem 40020 0
[1:t80018] EOF
[1:t40020] Master: Moj identyfikator to 262176
[1:t40020] Master: proces t40021 jest na hoscie lab-143-1
[1:t40020] Master: proces tc0019 jest na hoscie lab-143-3
[1:t40020] Master: proces t80018 jest na hoscie lab-143-2
[1:t40020] Master: proces t80019 jest na hoscie lab-143-2
[1:t40020] EOF
[1:t80019] Slave: Otrzymałem 40020 1
[1:t80019] EOF
[1] finished
```

Dla celów czysto edukacyjnych zmodyfikujemy teraz nieco programy master.c oraz slave.c. Mianowicie w pliku master.c wywołanie funkcji pvm_initsend z linijki 13 przeniesiemy przed pętlę for. Odpowiedni fragment kodu wyglądać będzie więc tak:

```
11.     pvm_initsend(PvmDataDefault);
12.     for( i=0 ; i<nproc ; i++ )
13.     {
14.           pvm_pkint(&mytid, 1, 1);
15.           pvm_pkint(&i, 1, 1);
16.           pvm_send(tids[i], MSG_MSTR);
17.     }
```

Wydawałoby się na pierwszy rzut oka nieuważnej osoby, że zmiana ta jest dość niewielka. Wprowadźmy więc równie niewielką zmianę w kodzie programu *slave*.

Zmiany te to wprowadzenie definicji trzech potrzebnych zmiennych w linijce 6 oraz dodanie wywołania funkcji `pvm_bufinfo` w linijce 13, za pomocą której uzyskamy informacje o rozmiarze odebranego komunikatu.

```
1.   int main()
2.   {
3.      int ptid, my2;
4.      int mytid;
5.      char *str = malloc( 64 );
6.      int size,tag,tid;
7.      int i = pvm_recv( -1, MSG_MSTR);
8.      if ( i < 0 )
9.      {
10.           perror("Nieoczekiwany błąd.");
11.           exit(0);
12.     }
13.     pvm_bufinfo( i, &size, &tag, &tid );
14.     printf("Slave: Rozmiar wiadomosci %d\n",size);
```

Po skompilowaniu oraz uruchomieniu programu (w przykładzie dla zwiększenia czytelności pominięto część informacji wypisywanych przez program *slave*), wynik może wyglądać tak:

```
pvm> spawn -> master
[1]
1 successful
tc001b
pvm> [1:t40026] Slave: Rozmiar wiadomosci 8
[1:t40026] EOF
[1:t40027] Slave: Rozmiar wiadomosci 16
[1:t40027] EOF
[1:tc001c] Slave: Rozmiar wiadomosci 32
[1:tc001c] EOF
[1:tc001b] Master: Moj identyfikator to 786459
[1:tc001b] Master: proces tc001c jest na hoscie lab-143-3
[1:tc001b] Master: proces t40026 jest na hoscie lab-143-1
[1:tc001b] Master: proces t40027 jest na hoscie lab-143-1
[1:tc001b] Master: proces t8001c jest na hoscie lab-143-2
[1:t8001c] Slave: Rozmiar wiadomosci 24
[1:tc001b] EOF
[1:t8001c] EOF
[1] finished
```

Widać tutaj wyraźnie, że każdy kolejny proces *slave* otrzymuje coraz większy komunikat (rozmiary kolejno to 8, 16, 24, 32). Wynika to z faktu, że ponieważ przenieśliśmy funkcję `pvm_initsend` przed pętlę for, bufor komunikacyjny jest tworzony tylko raz, a następnie nie jest czyszczony. Każda iteracja pętli powoduje dołożenie dwóch kolejnych elementów typu `int`, których rozmiar wynosi 4 bajty.

Przykład wykorzystania pozostałych argumentów funkcji `pvm_pkint`

Funkcja `pvm_pkint` posiada trzy argumenty. Do tej pory wartość dwóch ostatnich z nich zawsze wynosiła 1. Obecnie przedstawimy sposób użycia tych argumentów.

Przypomnijmy, że pierwszy argument to adres, spod którego mają być pobierane elementy do umieszczenia w buforze, drugi argument oznacza liczbę tych elementów, a trzeci tzw. *odstęp*. Przeanalizujmy kod przedstawiony poniżej:

`Program master.c`

```
1.    #include "def.h"
2.    #include <stdio.h>

3.    int main(int argc, char **argv)
4.    {
5.            int tids[SLAVENUM];
6.            int i;
7.            int tab[10]={0,1,2,3,4,5,6,7,8,9};
8.            pvm_spawn(SLAVENAME,0,0,".",SLAVENUM,tids);
```

W linijce 7-mej znajduje się definicja tablicy 10-elementowej. Każdy z elementów tej tablicy to kolejny numer od 0 do 9 – czyli element drugi ma wartość 3, element trzeci ma wartość 2 i tak dalej. Ułatwi to później zrozumienia sensu trzeciego argumentu funkcji `pvm_pkint`. Proces *master* tworzy następnie `SLAVENUM` procesów potomnych.

```
9.      for (i=0;i<SLAVENUM;i+=2)
10.     {
11.           pvm_initsend( PvmDataRaw );
12.           pvm_pkint(tab,5,2);
13.           pvm_send(tids[i],MSG_MSTR);
14.     }
```

Zobacz teraz co się dzieje w pętli w linijkach 9-14. Do co drugiego procesu (`i+=2`) procesu wysyłana jest teraz część tablicy `tab` (liczba elementów wynosi pięć) – odstęp jednak (trzeci argument funkcji `pvm_pkint`) wynosi 2. W efekcie wysyłane są *parzyste* elementy tablicy `tab`: 0, 2, 4, 6 i 8.

```
15.     for (i=1;i<SLAVENUM;i+=2)
16.     {
17.           pvm_initsend( PvmDataRaw );
18.           pvm_pkint(tab+1,5,2);
19.           pvm_send(tids[i],MSG_MSTR);
20.     }
21.     pvm_exit();
22.   }
```

W kolejnej pętli do co drugiego procesu (poczynając od `tids[1]`) wysyłanych jest pięć *nieparzystych* elementów tablicy `tab` (1,3,5,7,9).

Proces *slave* odbiera wysłane do siebie elementy i wypisuje je na ekranie:

`Program slave.c`

```
1.    #include "def.h"
2.    #include <stdio.h>
3.    int main()
4.    {
5.           int i;
6.           int tab[5];
7.           pvm_recv(-1,MSG_MSTR);
8.           pvm_upkint( tab, 5, 1);
9.           printf("Slave: ");
10.          for (i=0;i<5;i++)
11.          {
12.                printf("tab[%d]=%d ", i, tab[i]);
13.          }
14.          printf("\n");
15.          pvm_exit();
16.   }
```

Zauważ, że w linijce 8-mej trzeci argument funkcji `pvm_upkint` wynosi 1. Chcemy wypakować *wszystkie* pięć przysłanych elementów, a nie tylko co drugi z nich.

Należy teraz skompilować oba programy i umieścić pliki wykonywalne w odpowiednim katalogu:

```
gcc master.c –o master –lpvm3
gcc slave.c –o slave –lpvm3
cp master slave $PVM_HOME
```

Następnie można je 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
t4000a
pvm> [1:t4000a] EOF
[1:t80005] Slave: tab[0]=0 tab[1]=2 tab[2]=4 tab[3]=6 tab[4]=8
[1:tc0005] Slave: tab[0]=1 tab[1]=3 tab[2]=5 tab[3]=7 tab[4]=9
[1:t100005] Slave: tab[0]=0 tab[1]=2 tab[2]=4 tab[3]=6 tab[4]=8
[1:t80005] EOF
[1:t140006] Slave: tab[0]=1 tab[1]=3 tab[2]=5 tab[3]=7 tab[4]=9
[1:tc0005] EOF
[1:t100005] EOF
[1:t140006] EOF
[1:t4000b] Slave: tab[0]=0 tab[1]=2 tab[2]=4 tab[3]=6 tab[4]=8
[1:t4000b] EOF
[1] finished
```

Po zakończeniu ćwiczenia nie zapomnij zatrzymać środowisko PVM za pomocą komendy konsoli `halt`.

Poznane funkcje biblioteki PVM

`int bufid = pvm_initsend(int encoding)`

Jeśli użytkownik używa tylko jednego bufora, co jest najczęstszym przypadkiem, funkcja `pvm_initsend` jest jedyną potrzebną do zarządzania buforami. Funkcja ta czyści dotychczasowy bufor nadawczy i tworzy nowy dla kolejnego komunikatu. Zwracany jest identyfikator nowego bufora. Możliwe jest zdefiniowanie kodowania i dekodowania używanego przy przesyłaniu komunikatów pomiędzy potencjalnie heterogenicznymi węzłami. Możliwe są trzy warianty parametru `encoding`:

- `PvmDataDefault`: używana jest konwersja w standardzie XDR. Operacje te są nadmiarowe w przypadku wysyłania komunikatu na maszynę o takim samym formacie danych.
- `PvmDataRaw`: nie jest wykonywana żadna konwersja. Komunikat przesyłany jest w takiej samej postaci jak został umieszczony w buforze. Może to skutkować niewielką poprawą wydajności. Należy tej opcji używać wtedy, gdy posiada się pewność, że adresat wiadomości znajduje się na maszynie o identycznym formacie danych.
- `PvmDataInPlace`: przesyłane dane pozostają na miejscu w celu zmniejszenia narzutu czasowego spowodowanego kopiowaniem i konwersją danych. Bufor zawiera tylko rozmiar i wskaźnik do przesyłanych danych. Przy wywołaniu funkcji `pvm_send` dane pobierane są bezpośrednio z pamięci użytkownika.

`int errno = pvm_pkint(int &element, int no, int stride)`

`int errno = pvm_upkint(int &element, int no, int stride)`

`...`

Każda z powyżej wymienianych funkcji umieszcza w aktywnym buforze nadawczym tablicę wartości o określonym typie, bądź kopiuje tablicę z bufora do wskazywanego przez programistę obszaru pamięci. Należy zaznaczyć, że komunikat może się składać z wielu tablic różnego typu. Poniższa tabela przedstawia typ danych, odpowiadającą mu funkcję umieszczającą dane w buforze i wyczytujące je z niego.

Tabela 1 Funkcje do pakowania i wypakowywania danych z bufora komunikacyjnego

Typ danych	*Funkcja pakująca*	*Funkcja rozpakowująca*
byte	pvm_pkbyte	pvm_upkbyte
complex	pvm_pkcplx	pvm_upkcplx
double complex	pvm_pkdcplx	pvm_upkdcplx
double	pvm_pkdouble	pvm_upkdouble
float	pvm_pkfloat	pvm_upkfloat
int	pvm_pkint	pvm_upkint
long	pvm_pklong	pvm_upklong
short	pvm_pkshort	pvm_upkshort
string	pvm_pkstr	pvm_upkstr

PVM dostarcza również funkcję pakującą używającą formatu funkcji printf do określenia typu umieszczanych w buforze danych.

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

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

Funkcja pvm_send dodaje do danych w buforze nadawczym etykietę numeryczną `tag` i wysyła je do zadania z identyfikatorem `tid`. Różnica w przypadku polecenia pvm_mcast polega na tym, że po dodaniu etykiety komunikat wysłany zostaje do wszystkich zadań wyszczególnionych w tablicy tids.

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

Jest to funkcja powodująca blokujący odbiór komunikatu, tzn. sterowanie wraca do procesu dopiero po umieszczeniu komunikatu w buforze odbiorczym. Możliwe jest filtrowanie odbieranych komunikatów według identyfikatora nadawcy `tid` i/lub etykiety `msgtag`. Wartość -1 oznacza wszystkie możliwe wartości. Odebrany komunikat znajduje się w nowoutworzonym aktywnym buforze odbiorczym. Poprzedni aktywny bufor odbiorczy jest zwalniany, chyba, że został zapamiętany poleceniem `pvm_setrbuf`.

```
int tid = pvm_mytid(void)
```

Funkcja ta zwraca identyfikator `tid` wywołującego ją procesu i może być wywoływana wielokrotnie. Przy czym przy pierwszym wywołaniu funkcja rejestruje proces w maszynie wirtualnej nadając mu unikalny identyfikator.

```
int tid = pvm_parent (void)
```

Funkcja `pvm_parent` zwraca `tid` procesu, który wywołał polecenie `pvm_spawn` uruchamiające proces wywołujący `pvm_parent`. Możliwe jest zwrócenie stałej `PvmNoParent` jeśli proces nie był utworzony poleceniem `pvm_spawn`.

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

Funkcja `pvm_bufinfo` zwraca etykietę, identyfikator nadawcy i rozmiar w bajtach komunikatu znajdującego się w wyspecyfikowanym buforze odbiorczym. Funkcja ta jest pomoca przy określaniu pochodzenia komunikatu odebranego przy użyciu tzw. „dzikiej karty” (czyli któregoś z parametrów funkcji `pvm_recv` ustawionego na –1).

Zadanie do samodzielnego wykonania

W opisie funkcji biblioteki PVM zwróciłeś na pewno uwagę na dwie, których nie wykorzystywaliśmy dotąd w ćwiczeniu: `pvm_mcast` oraz `pvm_parent`. Samodzielnie wskaż miejsca, gdzie w napisanym przez nas programie można wykorzystać te funkcje. Zmodyfikuj oba programy (zarówno *slave* jak i *master*) oraz skompiluj je i uruchom.

Gotowe przykłady demonstrujące rozwiązanie znajdują się wśród materiałów kursu, ale nie zaglądaj do nich, dopóki samemu nie wykonasz tego zadania.

Wreszcie, w celu uzyskania większego obycia z biblioteką PVM, poeksperymentuj z możliwością filtrowania wiadomości podczas odbioru, ustawiając odpowiednio parametry (filtry) określającego możliwych nadawców i możliwy typ odbieranych komunikatów.

Podsumowanie

W wyniku tego ćwiczenia stworzyłeś pierwszą aplikację, której poszczególne elementy komunikowały się między sobą za pomocą środowiska PVM.

Co powinieneś wiedzieć:

- W jaki sposób następuje utworzenie i wysłanie komunikatów za pomocą funkcji bibliotecznych PVM (`pvm_initsend`, `pvm_pkint`, `pvm_pkstr` , `pvm_send`)
- Jakie funkcje biblioteczne służą do blokującego odbierania komunikatów (`pvm_recv`) i wyciągania ich zawartości (`pvm_upkint, pvm_upkstr`).
- Co to są i jaką pełnią funkcję znaczniki wiadomości (drugi parametr funkcji `pvm_send` oraz `pvm_recv`)
- W jaki sposób pobierać informację o własnym identyfikatorze zadania (`pvm_mytid`) oraz identyfikatorze zadania ojca (`pvm_parent`)