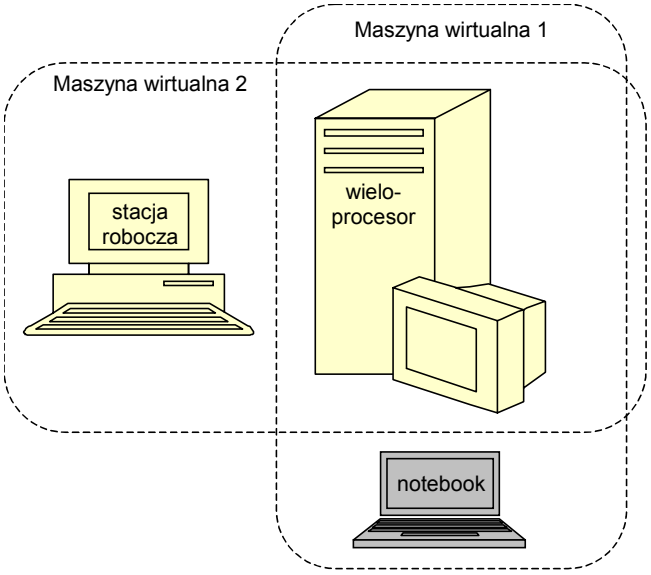
Środowiska przetwarzania rozproszonego

1 Parallel Virtual Machine

PVM (ang. Parallel Virtual Machine) powstał w 1989 roku w Oak Ridge National Laboratory, w celu ułatwienia programowania równoległego w środowisku komputerów heterogenicznych połączonych siecią. Idea PVM bazuje na pojęciu *maszyny wirtualnej*, czyli pewnej dodatkowej warstwy oprogramowania, na poziomie której poszczególne *węzły* systemu (ang. hosts) postrzegane są jako jednostki przetwarzające, a cała maszyna wirtualna jest postrzegana jako komputer równoległy z rozproszoną pamięcią.

Maszyn wirtualna konfigurowana jest dla poszczególnych użytkowników, co oznacza, że ten sam węzeł może być składnikiem wielu maszyn wirtualnych. Inaczej rzecz ujmując, różni użytkownicy tego samego komputera (hosta) mogą jednocześnie włączyć go do swoich maszyn wirtualnych (Rysunek 1). Podstawowym elementem konstrukcyjnym maszyny wirtualnej jest demon systemowy *pvmd*. Każdej maszynie wirtualnej, w skład której wchodzi dany węzeł, odpowiada jeden demon.



Rysunek 1 Maszyna wirtualna

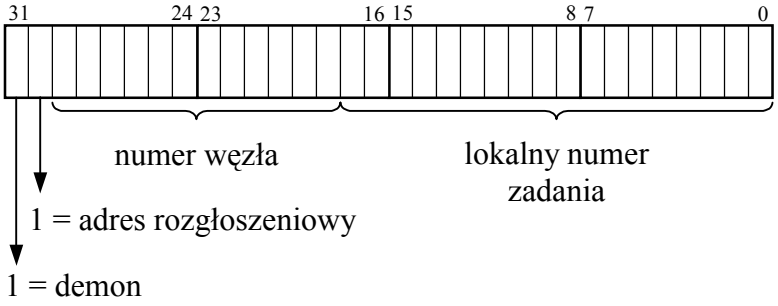
Przetwarzanie w środowisku PVM polega na wykonywaniu programów tworzących aplikację równoległą na poszczególnych węzłach maszyny wirtualnej. Na poziomie maszyny wirtualnej elementarną jednostką reprezentującą wykonywany program jest *zadanie* (ang. task). Każde zadanie jest też procesem w lokalnym systemie operacyjnym węzła, a jego praca jako zadania PVM jest kontrolowana przez lokalny demon w tym węźle. Ponieważ lokalny demon również jest zadaniem PVM, zadania realizujące program aplikacji będą nazywane zadaniami użytkowymi.

Podstawową funkcją środowiska PVM jest ułatwianie uruchamiania zadań na poszczególnych węzłach maszyny wirtualnej oraz dostarczanie mechanizmów do komunikacji, synchronizacji i identyfikacji tych zadań niezależnie od ich fizycznej lokalizacji na poszczególnych węzłach. Ponadto, PVM umożliwia dynamiczną zmianę konfiguracji maszyny wirtualnej oraz udostępnia mechanizm dynamicznych grup procesów, ułatwiających implementację algorytmów równoległych. Skorzystanie z funkcjonalności PVM wymaga użycia w programach zadań funkcji PVM, których implementacja jest dołączana na etapie konsolidacji w postaci biblioteki *libpvm3.a* i *libgpvm3.a* dla programów w języku C lub *libfpvm3.a* dla programów w języku Fortran.

1.1 Wybrane elementy konstrukcji środowiska PVM

Większość decyzji projektowych podjętych przez twórców środowiska PVM wynikała z założenia, że powinno być ono dostępne na możliwie dużej liczbie platform sprzętowych i systemowych. Podstawowym systemem operacyjnym, na którym działał PVM był UNIX w niemal wszystkich jego odmianach. Z czasem pojawiły się realizacja na VMS oraz Microsoft Windows. Pewne rozwiązania przyjęto zatem z względu na popularność i szeroką dostępność wykorzystanych mechanizmów, często koszem efektywności.

Kluczowym elementem w systemie rozproszonym jest komunikacja pomiędzy zadaniami, która wymaga z kolei mechanizmu jednoznacznej identyfikacji zadań oraz ich fizycznej lokalizacji w systemie. W PVM zadania identyfikowane są przez 32-bitową wartość liczbową, zwaną *TID* (ang. task identifier), nadawaną przez lokalny demon. Wartość tego identyfikatora składa się z czterech pól, zawierających odpowiednio: numer węzła, na którym działa zadanie, numer zadania na tym węźle, bit identyfikujący zadanie-demon i bit określający adres rozgłoszeniowy (ang. multicast address). Format identyfikator przedstawia Rysunek 2



Rysunek 2 Identyfikator zadania

W środowisku PVM w ogólności przyjmuje się, że zadania nie mają możliwości współdzielenia jakiegokolwiek obszaru pamięci, zatem kooperacja między nimi odbywa się tylko poprzez przekazywanie komunikatów (ang. message passing). Komunikat identyfikowany jest przez TID nadawcy i przez *etykietę*, tj. 16-bitową liczbę całkowitą. Model komunikacyjny przyjęty w PVM zakłada, że każde zadanie może wysyłać komunikaty do każdego innego, oraz że nie istnieją ograniczenia na rozmiar i ilość komunikatów. Jedynym ograniczeniem jest fizyczny rozmiar pamięci na poszczególnych węzłach maszyny wirtualnej.

PVM zapewnia *komunikację asynchroniczną*, co oznacza, że po wysłaniu komunikatu zadanie-nadawca nie czeka, aż komunikat dotrze do odbiorcy, tylko natychmiast kontynuuje swoje przetwarzanie. Odpowiedzialność za przekazanie komunikatu przejmują odpowiednie demony. Maszyna wirtualna gwarantuje jedynie zachowanie lokalnego porządku (porządku FIFO), tzn. jeśli jedno zadanie wysyła dwa komunikatu do innego zadania, to komunikaty zostaną doręczone w kolejności, w jakiej zostały nadane. Adresatem komunikatu może być pojedynczy proces identyfikowany przez konkretny TID, może to być zbiór procesów, których identyfikatory przekazywane są w odpowiedniej strukturze, lub może to być dynamiczna grupa procesów identyfikowanych przez nazwę.

Odbiór komunikatu polega na jego wyciągnięciu z bufora po stronie odbiorczej, przy czym możliwe są dwie formy odbioru: *blokująca* i *nieblokująca*. Odbiór blokujący oznacza, że zadanie odbierające zostaje zawieszone do momentu dotarcia komunikatu, jeśli nie dotarł on wcześniej. W przypadku odbioru nieblokującego funkcja zwraca sterowanie natychmiast, przekazując odpowiedni status zależnie od tego, czy komunikat dotarł czy nie.

Komunikacja pomiędzy zadaniami najczęściej odbywa się za pośrednictwem demonów, które kontrolują pracę tych zadań. W komunikacji pomiędzy demonami wykorzystywany jest protokół UDP, a komunikacja pomiędzy zadaniem a lokalnym demonem odbywa się na

protokole TCP. Użycie protokołu UDP w komunikacji pomiędzy demonami daję większą elastyczność, ale ze względu na zawodność wymaga dobudowania dodatkowej funkcjonalności w postaci mechanizmu potwierdzeń i retransmisji.

Poszczególne zadania aplikacyjne uruchamiane są przez odpowiednie demony. W czasie konfiguracji maszyny wirtualnej muszą zostać również zdalnie uruchomione same demony. W tym celu wykorzystywana jest usługa *rsh* (remote shell), umożliwiająca zdalne wykonanie polecenia bez potrzeby logowania użytkownika w trybie interaktywnym. Po odpowiednim skonfigurowaniu pliku `~/.rhosts` uruchamianie zdalnych poleceń możne się odbywać bez podawania hasła.

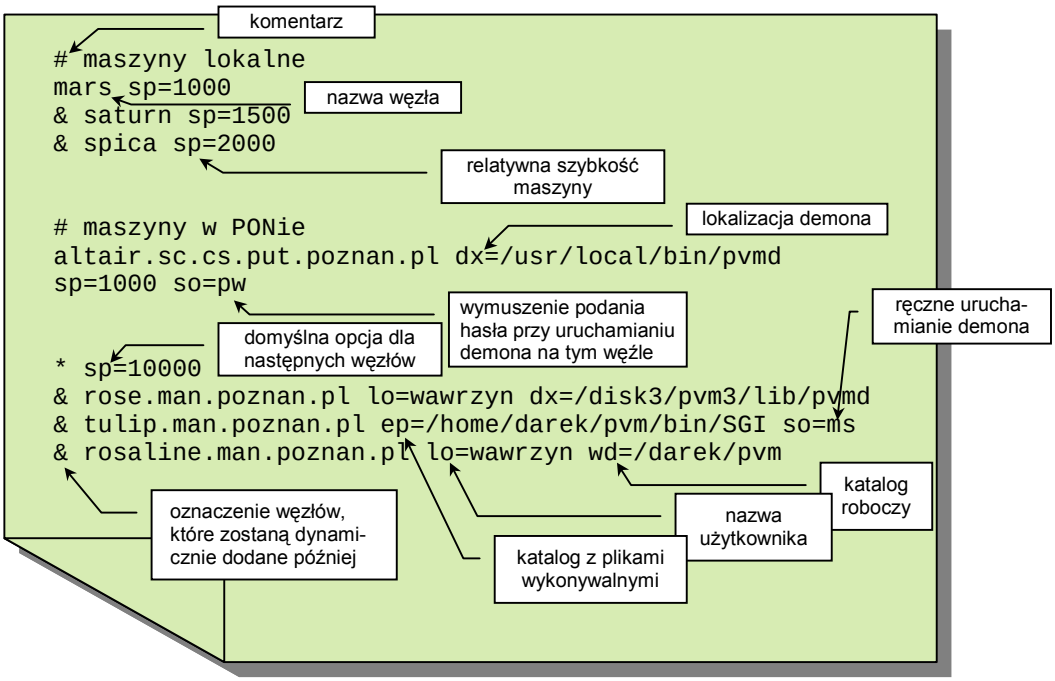
Elementem ułatwiającym programowanie równoległe poprzez możliwość dynamicznej dekompozycji instancji problemu jest mechanizm grupowania zadań. Zarządzanie grupami zadań w środowisku PVM jest całkowicie scentralizowane i spoczywa na serwerze grup — *pvmgs*. Jest to specjalne zadanie systemowe, uruchamiane automatycznie na każdej maszynie wirtualnej — dokładniej na pierwszym węźle tej maszyny — w momencie utworzenia pierwszej grupy procesów. Z zadaniem tym komunikują się inne zadania lub demony w czasie wykonywania funkcji grupowych. Rozwiązanie scentralizowane ma istotną wadę — istnieje ryzyko powstania wąskiego gardła w przypadku intensywnego korzystania z mechanizmu grup przez zadania działające na maszynie wirtualnej.

1.2 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 rozdziale 1.3.
   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 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. W celu prawidłowego skonfigurowania maszyny wirtualnej pierwszy z uruchamianych demonów powinien dostać plik z opisem węzłów. Postać takiego pliku przedstawia Rysunek 3



Rysunek 3 Plik z opisem węzłów

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 staję 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 UNIX-ie, 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.

1.3 Funkcje biblioteczne PVM

Funkcje biblioteczne PVM zostały podzielone na następujące grupy, zgodnie z zakresem ich stosowalności. Wyróżnione zostały następujące grupy: obsługa zadań, konfiguracja maszyny wirtualnej, komunikacja pomiędzy zadaniami, dynamiczne grupy procesów i powiadamianie.

1.3.1 Obsługa zadań

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

Funkcja ta zwraca *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 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.

```
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. Argument `where` pozwala uruchomić zadania na konkretnym węźle, na konkretnej architekturze, lub pozostawia wybór miejsca systemowi w celu równoważenia obciążenia maszyny wirtualnej.

```
int info = pvm_kill(int tid)
```

Funkcja `pvm_kill` powoduje zakończenie zadania identyfikowanego przez `tid`. Nie zaleca się kończenia w ten sposób zadania wywołującego tę funkcję.

```
int info = pvm_task(int which, int *ntask, struct pvmtaskinfo
           **taskp)
```

Funkcja `pvm_task` zwraca informację o zadaniach PVM wykonywanych aktualnie na maszynie wirtualnej. Parametr `which` decyduje o tym czy interesują nas zadania z całej maszyny wirtualnej, czy tylko te pracujące na konkretnym węźle. Ilość zadań zwracana jest przez zmienną `ntask`. Każda struktura `pvmtaskinfo` zawiera informacje o identyfikatorze zadania, identyfikatorze demona, rodzicu, statusie zadania

i nazwie pliku wykonywalnego. W przypadku zadań uruchomionych z linii poleceń ta ostatnia pozycja pozostaje pusta.

```
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_catchout(FILE *ff)
```

Domyślnie PVM zapisuje standardowe wyście *stdout* i wyjście diagnostyczne *stderr* do specjalnego pliku zwanego log'iem o lokalizacji `/tmp/pvml/`<*uid*>. Funkcja `pvm_catchout` powoduje przechwytywanie wyjścia procesów uruchomionych za pomocą `pvm_spawn`. Znaki umieszczane w strumieniach *stdout* i *stderr* przez zadania „dzieci" zbierane są i przesyłane w komunikatach kontrolnych do procesu rodzica, gdzie każda linia opatrzona zostaje identyfikatorem procesu i wyświetlona na standardowym wyjściu. W przypadku wywołania funkcji `pvm_exit` przez proces rodzica przechwytującego wyjścia dzieci proces ten zostanie wstrzymany aż do chwili, gdy wszystkie procesy potomne opuszczą maszynę wirtualną.

1.3.2 Konfiguracja maszyny wirtualnej

```
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. `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.

```
int dtid = pvm_tidtohost(int tid)
```

Funkcja `pvm_tidtohost` zwraca identyfikator demona maszyny wirtualnej działającego na tym samym węźle co zadanie `tid`.

1.3.3 Przesyłanie komunikatów

Wysłanie komunikatu składa się w PVM z trzech kroków:

1. Konieczna jest inicjalizacja bufora za pomocą funkcji `pvm_initsend`, lub `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.

1.3.3.1 Bufory komunikacyjne

```
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.

`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.

Pozostałe omawiane tu funkcje wymagane są jedynie w przypadku, gdy użytkownik zarządza wieloma buforami wewnątrz pojedynczej aplikacji. W przypadku PVM w każdej chwili istnieje jeden aktywny bufor nadawczy i jeden aktywny odbiorczy. Użytkownik może jednak stworzyć wiele buforów i później przełączać się pomiędzy nimi.

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

Funkcja ta tworzy nowy pusty bufor wysyłający i ustawia dla niego metodę konwersji. Zwracany jest identyfikator bufora.

```
int info = freebuf(int bufid)
```

Funkcja ta zwalnia bufor o identyfikatorze `bufid`. Powinna być wywołana po tym jak komunikat został wysłany i nie jest już więcej potrzebny.

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

Funkcje te zwracają identyfikator aktywnego bufora nadawczego lub odbiorczego.

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

Funkcje te ustawiają aktywny bufor nadawczy bądź odbiorczy, zapamiętują stan poprzedniego bufora i zwracają jego identyfikator. Jeżeli argumentem funkcji jest 0 to zapamiętywany jest poprzedni bufor i aplikacja nie ma aktywnego bufora.

1.3.3.2 Umieszczanie danych w buforze i ich odbiór

Każda z poniżej 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.

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.

1.3.3.3 Wysyłanie i odbieranie komunikatów

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

Funkcja `pvm_psend` umieszcza w buforze i wysyła tablicę danych o podanym typie do zadania o identyfikatorze `tid`.

```
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 i/lub etykiety. 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 bufid = pvm_nrecv(int tid, int tag)
```

Funkcja analogiczna do powyższej, z tą tylko różnicą , że jest nieblokująca. Jeśli nie ma komunikatu, który mógłby zostać odebrany zwracana jest wartość 0.

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

Jeśli żądany komunikat nie nadszedł `pvm_probe` zwraca 0. W przeciwnym przypadku zwraca identyfikator bufora, ale nie odbiera komunikatu. Funkcja ta wykorzystywana jest do określania czy komunikat już nadszedł.

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

PVM dostarcza również funkcję blokującego odbiory zaopatrzoną w mechanizm timeout-u. Czekanie na nadejście komunikatu ogranicza się tylko do zdefiniowanego przez użytkownika periodu czasowego, po którym zwracana jest informacja o braku komunikatu o ile ten wcześniej nie nadszedł. Jeśli użytkownik zdefiniuje długi czas oczekiwania na komunikat funkcja ta upodabnia się do `pvm_recv`. Jeśli czas będzie krótki bądź 0 to możemy mówić o funkcjonalności zbliżonej do polecenia `pvm_nrecv`. Jak widać jest to więc funkcja pośrednia pomiędzy blokującym a nieblokującym odbiorem.

```
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 „dzikiej karty”.

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

Funkcja `pvm_precv` łączy w sobie funkcjonalność funkcji blokującego odbierania i rozpakowania komunikatu. Funkcja ta nie zwraca identyfikatora bufora, zamiast tego zwracany jest identyfikator nadawcy, etykieta i rozmiar komunikatu.

```
int (*old)() = pvm_recvf(int (*new)(int buf, int tid, int tag))
```

Funkcja ta służy do zmiany funkcji zatwierdzającej komunikat do odbioru. Standardowo sprawdzany jest nadawca i etykieta.

1.3.4 Dynamiczne grupy procesów

W PVM grupy są dynamiczne, co znaczy, że dowolne zadanie może w każdej chwili przyłączyć się do grupy, bądź ją opuścić 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` i `pvm_reduce`, które ze względu na swoją specyfikę wymagają, aby wywołujące je zadanie było członkiem grupy.

```
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 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.

```
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 *broadcastu* może spowodować, że komunikat nie zostanie przez ten proces odebrany. Podobnie jeśli proces opuści grupę po rozpoczęciu operacji *broadcastu* i tak otrzyma komunikat.

1.3.5 Powiadamianie

```
int info = pvm_sendsig(int tid, int signum)
int info = pvm_notify(int what, int msgtag, int cnt, int tids)
```

Funkcja `pvm_sendsig` wysyła sygnał `signum` systemu UNIX do innego zadania PVM o identyfikatorze `tid`. Funkcja `pvm_notify` żąda od PVM powiadamiania zadania wywołującego w przypadku zajścia określonego zdarzenia, którym może być:

- opuszczenie maszyny wirtualnej przez wyspecyfikowane zadanie,
- usunięcie lub awaria węzła,
- dodanie nowego węzła obliczeniowego.

W przypadku zajścia zarejestrowanego zdarzenia do procesu wysyłany jest komunikat o etykiecie `msgtag` zawierający opis zdarzenia.

1.4 Konsola PVM

Konsola PVM jest zadaniem systemowym, które udostępnia wybrane polecenia PVM w trybie interaktywnym, tzn. można wydać polecenie interpretowane przez konsolę PVM, i natychmiast zobaczyć wyniki. Konsola udostępnia głownie funkcje związane z obsługą zadań i konfiguracją maszyny wirtualnej i najczęściej wykorzystywana jest do dynamicznej zmiany konfiguracji maszyny wirtualnej oraz do kontroli pracy zadań. Najczęściej wykorzystywane polecenia zatem to:

- `add` — dodawanie węzła do maszyny wirtualnej,
- `delete` — uwuwanie węzła z maszyny wirtualnej,
- `conf` — sprawdzanie bieżącej konfiguracji maszyny wirtualnej,
- `ps` — wyświetlanie listy zadań.