Detekcja zakończenia i obraz stanu globalnego

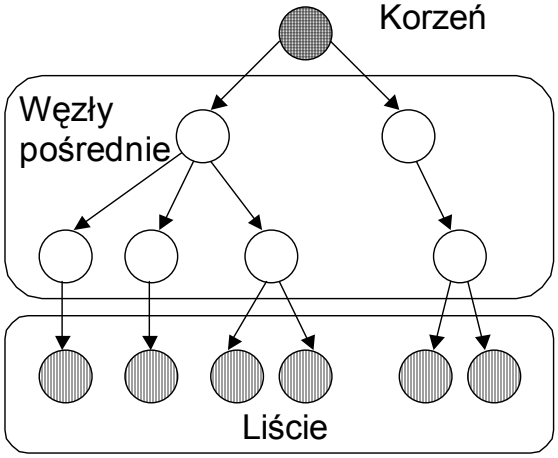
Zakres ćwiczenia

Celem ćwiczenia jest utrwalenie umiejętności zdobytych w czasie poprzednich zajęć oraz zwiększenie znajomości funkcji biblioteki PVM.

Przedstawienie problemu

Są to ostatnie ćwiczenia z biblioteką PVM. Poznasz tylko jedną prostą funkcję, której znajomość nie będzie jednak niezbędna do wykonania zadania. Napiszesz jeden, za to duży i skomplikowany program. Zadanie polegać będzie na utworzeniu drzewa procesów a następnie rozesłania wiadomości o **wszystkich** procesach do **wszystkich** procesów. Założymy, że żaden proces nie posiada żadnych informacji na temat ile procesów poza nim istnieje w systemie. Zakładamy, że jedyne informacje dostępna każdemu procesowi to liczba (i identyfikatory) utworzonych przez niego potomków. Dodatkowo przyjmiemy, że liczba poziomów drzewa jest ustalona z góry (choć nieznana), a każdy proces wie, na którym poziomie drzewa się znajduje.

Utworzona struktura wyglądać będzie więc następująco:



Rysunek 1 Drzewo procesów

W rzeczywistości dla ułatwienia analizy wykonania programu każdy proces będzie tworzył tyle samo potomków, ale zakładamy, że ta wiedza jest niedostępna dla procesów.

Jak jest to przedstawione na rysunku 1, procesy można podzielić na trzy rodzaje: liście, węzły pośrednie oraz korzeń drzewa, którym jest przodek wszystkich pozostałych. Od ciebie zależy, czy będziesz chciał wykorzystać tą informację. W materiałach kursu znajdziesz jedno z możliwych, ale to wcale nie znaczy że najlepsze, rozwiązanie.

Podsumowując, twoim zadaniem będzie więc zapewnienie, by każdy proces (nie tylko korzeń czy inny wyróżniony proces) należący do drzewa znał liczbę i identyfikatory wszystkich pozostałych procesów.

Tworzenie drzewa procesów

Obecnie przedstawimy sposób tworzenia drzewa procesów. Sposobu tego nie powinieneś modyfikować. Oczywiście musisz pamiętać o zadeklarowaniu i zainicjowaniu wszystkich używanych w poniższych fragmentach kodu zmiennych. Nie powinno to dla ciebie być problemem. Jeżeli jest, powinieneś powrócić do poprzednich ćwiczeń.

Program master.c

```
1. maxind=pvm_spawn(SLAVENAME,0,0,".",SLAVENUM,tids);
2. pvm_initsend( PvmDataDefault );
3. pvm_pkint(&ile_poziomów_drzewa,1,1);
4. pvm_mcast(tids,maxind,MSG_MSTR);
```

Proces *master*, korzeń tworzonego drzewa, tworzy `SLAVENUM` procesów. Następnie do wszystkich potomków przesyła informację, ile ma być maksymalnie utworzonych poziomów drzewa. Uwaga: zakładamy, że procesu **nie wiedzą**, że co ta liczba oznacza.

Program slave.c

```
1.   pvm_recv( -1, MSG_MSTR );
2.   pvm_upkint( &poziom, 1, 1);
3.   lisc=1;
4.   if (poziom)
5.   {
6.        int poziom_send = poziom;
7.        lisc=0;
8.        poziom_send--;
9.        pvm_spawn(SLAVENAME,0,0,".",SLAVENUM,tids);
10.       pvm_initsend( PvmDataDefault );
11.       pvm_pkint(&poziom_send,1,1);
12.       pvm_mcast(tids,nres,MSG_MSTR);
13.  }
```

Jak w oczywisty sposób wynika z analizy powyższego kodu, procesy *slave* traktują nadesłaną liczbę jako polecenie, czy tworzyć, czy też nie potomków. Jeżeli zmienna `poziom` wypakowana z komunikatu jest niezerowa, proces slave tworzy nowe procesy dekrementuje tą zmienną (zapamiętując oryginalną wartość) i rozsyła ja do utworzonych potomków.

Zmienna `lisc` określa, czy proces *slave* jest węzłem pośrednim (wartość `lisc == 0`) czy też liściem właśnie. Dzięki temu nie ma potrzeby tworzenia osobnego pliku – proces może wykonywać różne akcje w zależności od wartości tej zmiennej. Oczywiście jeżeli chcesz, możesz taki plik utworzyć i zmodyfikować powyższy kod, tak, by wyglądał w ten sposób:

```
8.        poziom_send--;
9.        if (poziom_send)
10.           pvm_spawn(SLAVENAME,0,0,".",SLAVENUM,tids);
11.       else
12.           pvm_spawn(LEAFNAME,0,0,".",SLAVENUM,tids);
```

Omówienie koncepcji rozwiązania

Większość studentów napotykając ten problem natychmiast wpada na jedno z dwóch nieprawidłowych rozwiązań (na szczęście, w większości przypadków orientują się, że rozwiązania te są nieprawidłowe jeszcze w trakcie ich opisywania). Naturalnym impulsem jest przekazanie procesom potomnym identyfikatora korzenia drzewa procesów, a następnie wymagania, by każdy z nich odesłał korzeniowi swój własny identyfikator. Po zebranie wszystkich odpowiedzi korzeń rozsyłałby je do wszystkich procesów. Oczywistym problemem jest jednak, skąd przodek ma wiedzieć, że posiada już wszystkie niezbędne informacje? Skąd ma wiedzieć, że rozpoznawanie drzewa procesów się skończyło i wszystkie procesy potomne przesłały już mu swoje identyfikatory?

Drugim, identycznym co do koncepcji a jedynie różnym w implementacji jest wykorzystanie grup. Studenci sugerują, by wszystkie procesy dołączyły do jednej grupy i następnie rozesłały informacje o

sobie do całej grupy – dzięki temu nie ma potrzeby przesyłania w dół identyfikatora przodka wszystkich procesów. Korzeń drzewa następnie po zebraniu wszystkich odpowiedzi i skonstruowaniu pełnej informacji o stanu grupy odesłałby te informacje do grupy. Pytanie oczywiście brzmi identycznie jak poprzednio: skąd proces *master* ma wiedzieć, że otrzymałby wszystkie odpowiedzi?

Oba te rozwiązania są scentralizowane. Rzadziej pojawia się sugestia rozwiązania zdecentralizowanego, będącego prostym rozwinięciem poprzedniej koncepcji. Każdy proces dołączałby do grupy i wysyłał do niej swój identyfikator, jak poprzednio. Każdy proces zbierałby te informacje konstruując obraz całego drzewa. Problem oczywiście się nie zmienia: różnica polega tylko na tym, że obecnie pytanie „czy otrzymano już wszystkie informacje" zadawałyby sobie wszystkie procesy, a nie tylko wyróżniony korzeń drzewa. Drugim pytaniem, które się tutaj pojawia, brzmi: jaka jest gwarancja, że procesy faktycznie otrzymają wszystkie rozwiązanie? W końcu, część procesów może rozpocząć rozsyłanie w chwili, w której nie wszystkie pozostałe procesy zdążyły dołączyć do grupy. Nie można zastosować tutaj jednej bariery w celu synchronizacji – bo nie wiadomo, ile procesów ma tę barierę wywołać. Można ten problem rozwiązać jedynie stosując dodatkowe grupy i dużo dodatkowych synchronizujących operacji barier.

Spróbuj samemu opracować rozwiązanie tego problemu, zanim zaczniesz czytać następne akapity.

Pierwszą sugestią jest rozsyłanie przez wszystkie procesy pary dwóch liczb: poziomu oraz liczby stworzonych potomków. Dla liście obydwie te liczby będą wynosiły zero. Proces master zbierając informacje będzie w stanie na ich podstawie wyliczyć liczbę wszystkich procesów. Zastanów się, dlaczego przesłanie informacji o poziomie również jest niezbędne?

Drugim rozwiązaniem jest wykorzystanie struktury drzewa. Nie jest to oczywiste na pierwszy rzut oka, ale oparty jest on (nie wprost) na algorytmie dla przetwarzania dyfuzyjnego przedstawionym na wykładzie. Każdy proces wie, ile wysłał wiadomości do potomków. Procesy-liście mogą przesyłać do procesów macierzystych informację o swoim identyfikatorze. Wszystkie procesy macierzyste oczekują od potomków na wiadomości dostarczające informacje o nich samych i wszystkich ich pośrednich i bezpośrednich potomkach – wiedzą, ile tych wiadomości powinno być, gdyż wiedzą, ilu utworzyły potomków. Po otrzymaniu wiadomości od wszystkich procesów, posiadają informacje na temat poddrzewa procesów, którego same są korzeniem; informacje te przesyłają z kolei do swojego rodzica. Korzeń drzewa po otrzymaniu wiadomości od wszystkich swoich potomków posiada informacje o wszystkich procesach i może już wtedy je rozesłać do nich wszystkich.

Trywialne jest rozwiązanie wtedy kolejnego problemu: powiadomienia procesów o chwili, w której algorytm się kończy. Wystarczy, by procesy odbierając informacje o stanie całego drzewa, wysyłały potwierdzenia (może już bezpośrednio do korzenia: tym razem korzeń już wie, ile jest wszystkich procesów, a więc wie, ile powinno być potwierdzeń). Po zebraniu wszystkich potwierdzeń proces-korzeń wysyła specjalną wiadomość zatwierdzającą. W momencie otrzymania każdy proces wie, że wszystkie inne procesy otrzymały już informacje na temat całego badanego drzewa. Drugim rozwiązaniem jest wykorzystanie grupy i operacji bariery.

Wskazówki do rozwiązania

W celach ułatwienia weryfikacji poprawności programów możesz użyć funkcji biblioteki PVM o nazwie `pvm_task`, która zwraca informacje o liczbie i identyfikatorach wszystkich procesów operujących na maszynie wirtualnej.

Problemem rozwiązywanym przez ciebie ma być zbieranie informacji, a nie to, jak to zaimplementować w języku C. Możesz więc wykorzystać następujący krótki fragment kodu:

```
1. pvm_upkint(&rozmiar,1,1);
2. pvm_upkint(&(alltids[maxind]),rozmiar,1);
3. maxind+=rozmiar;
```

W fragmencie powyżej, komunikat zawiera dwa pola: `rozmiar`, które jak łatwo się domyśleć określa liczbę przysyłanych identyfikatorów procesów, oraz właśnie identyfikatory. Są one wypakowywane do tablicy `alltids`, zaczynając wpisy od elementu o indeksie `maxind`. Zmienna `maxind` reprezentuje liczbę wszystkich nadesłanych do tej pory identyfikatorów, a więc musi być zsumowana z zmienną `rozmiar`. Dla ułatwienia, niech `alltids` będzie zwykłą tablicą o z góry ustalonym, maksymalnym rozmiarze (na przykład, 1000 elementów).

Wynik wywołania napisanego przez ciebie programu może wyglądać na przykład tak:

```
pvm> spawn -> master
[1]
1 successful
tc025e
pvm> [1:tc025e] libpvm [tc025e]: child task(s) still running.
waiting...
[1:tc025e] [tc025f] BEGIN
...
[1:tc025e] Wg. pvm_tasks procesow jest 46
[1:tc025e] tc025e tc025f tc0260 tc0261 tc0262 tc0263 tc0264
tc0265 tc0266 t14025e t14025f t140260 t140261 t140262 t140263
t140264 t140265 t140266 t10025e t10025f t100260 t100261 t100262
t100263 t100264 t100265 t100266 t40318 t40319 t4031a t4031b
t4031c t4031d t4031e t4031f t40320 t40321 t40322 t80260 t80261
t80262 t80263 t80264 t80265 t80266 t80267 t80268
[1:tc025e] Wszystkie procesy już wiedzą…. W systemie jest 46
procesów
[1:tc025e] t100262 tc0261 t100260 t100263 tc0263 t140261 t10025e
t80265 tc0262 t80261 t80266 t4031e tc0260 t14025e tc025f t140263
t4031f t140262 t140265 t40320 t100261 t14025f t100265 tc0264
t4031b t80267 t40321 t80263 t4031a t80260 t140264 t100264 t80262
t140266 t40322 t4031c t10025f t100266 tc0265 t80264 t80268 tc0266
t4031d t140260 t40319 tc025e
[1:tc025e] [tc025f] EOF
....
[1:tc025e] EOF
[1] finished
```

Poznane funkcje biblioteki PVM

```
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.

Podsumowanie

Jeżeli udało ci się napisać samodzielnie program rozwiązujący przedstawiony problem, to znaczy, że posiadasz już pewną intuicję na temat problemów konstrukcji stanu globalnego i detekcji zakończenia.

Na tym kończymy przegląd środowiska PVM. W następnym module zapoznasz się z możliwościami konkurencyjnego rozwiązania o nazwie MPI.

Co powinieneś wiedzieć:

- Ćwiczenie nie zawiera żadnych nowych ważnych informacji.