Zegary logiczne

Zakres ćwiczenia

W obecnym ćwiczeniu nie zostaną wprowadzone żadne nowe funkcje biblioteki PVM. Zobaczysz za to program demonstrujący zagrożenia niesione przez używanie zegarów fizycznych w środowisku rozproszonym oraz napiszesz własną implementację algorytmu Lamporta, poznanego na wykładzie. Celem jest więc zarówno zastosowanie w praktyce wiadomości z wykładu jak i przećwiczenie umiejętności zdobytych poprzednio.

Problem rozbieżności czasów

Jednym z wielu problemów pojawiających się w systemach rozproszonych jest rozbieżność wartości zegarów na poszczególnych węzłach-maszynach. Każda maszyna może posiadać zegar chodzący z inną prędkością, zegary mogą się śpieszyć bądź spóźniać. Oczywiście w wielu zastosowaniach te rozbieżności czasów nie są istotne; niemniej jednak istnieje bardzo dużo propozycji zmierzających do synchronizacji zegarów - takich jak algorytm Cristiana, algorytm synchronizacji zegarów Berkeley czy też protokoły NTP (ang. *Network Time Protocol*) albo DTS (ang. *Distributed Time Service*) - z których jedną jest stosowanie czasu logicznego czy też wirtualnego wyznaczanego za pomocą zegarów logicznych (ang. *logical clocks*). W ogólności, czas wirtualny zależy tylko od zdarzeń zachodzących w systemie, a nie od wartości zegarów fizycznych.

Istotą zegarów logicznych jest żądanie zapewnienia, by każde zdarzenie w systemie posiadało własną, unikalną etykietę czasową, a w szczególności by zdarzenie wysłania wiadomości miało zawsze etykietę czasową mniejszą niż zdarzenie odebrania wiadomości (by nie można było odbierać wiadomości „z przyszłości"). Spełnienie tych wymagań jest warunkiem koniecznym poprawności niektórych algorytmów (na przykład wielu algorytmów wzajemnego wykluczania).

Aby przekonać się, na czym polega problem, stworzymy program używający zwykłych zegarów fizycznych (czas rzeczywisty). Kolejnym krokiem będzie samodzielne stworzenie programu implementującego ideę zegarów logicznych (czas wirtualny).

Koncepcja programu

Zmodyfikujemy poprzednio wykorzystywane programy. Nasza aplikacja będzie się składała tak jak poprzednio z dwóch części: *mastera* oraz uruchomionych przez niego procesów typu *slave*. Każdy *slave* będzie wysyłał wiadomość do *mastera*. Wiadomość ta będzie zawierała etykietę czasową zawierającą czas wysłania wiadomości. Program *master* odbierając tą wiadomość będzie porównywał ją z swoim czasem lokalnym.

Ćwiczenie to oczywiście będzie miało sens tylko wówczas, jeżeli maszyna wirtualna faktycznie będzie się składała z kilku węzłów (na jednej maszynie oczywiście nie da się zaobserwować rozbieżności między zegarami – wszystkie zadania korzystają na niej z tego samego zegara). Najpierw uruchom więc konsolę PVM i dodaj kilka zdalnych węzłów do maszyny wirtualnej. Wyjdź z konsoli używając komendy `quit`.

Przygotowanie programu

Struktura programu będzie identyczna jak poprzednio. Będzie się on składać z trzech plików: `def.h`, `master.c`, `slave.c`. Pierwszy z tych plików będzie zawierał definicje wspólne dla programów *master* oraz *slave* i jego budowa jest identyczna jak w poprzednich ćwiczeniach.

Pierwsza linijka zawiera dyrektywę preprocesora `include` załączającą plik nagłówkowy `sys/time.h` (ewentualnie linijkę tę można przenieść do pliku `def.h`). Plik ten zawiera potrzebną dla nas definicję używanej w programie struktury `timeval` oraz funkcji `gettimeofday`. W linijce 9 pojawia się definicja zegarów używanych dalej, oraz zmiennych `clock_sec` oraz `clock_usec`, które będą przechowywać etykietę czasową nadchodzącej wiadomości (odpowiednio: liczbę sekund i

mikrosekund). W linijce 10 znajduje się definicja zmiennej `tv` typu strukturalnego `timeval`, którą będziemy używali do pobrania czasu rzeczywistego systemu.

`program master.c`

```
1.   #include <sys/time.h>
2.   #include "def.h"

3.   main()
4.   {
5.      int mytid;
6.      int tids[SLAVENUM];
7.      char slave_name[NAMESIZE];
8.      int nproc, i;
9.      long clock_sec, clock_usec;
10.     struct timeval tv;
11.
12.     mytid = pvm_mytid();
13.     nproc=pvm_spawn(SLAVENAME, NULL, PvmTaskDefault, \
                                    "", SLAVENUM, tids);
14.     if (nproc < SLAVENUM )
15.     {
16.           perror("Blad podczas uruchamiania");
17.           return -1;
18.     }
19.     printf("Master: Moj identyfikator to %d \n", mytid );

20.     for( i=0 ; i<nproc ; i++ )
21.     {
22.           pvm_recv( -1, MSG_SLV );
23.           gettimeofday(&tv,NULL);
24.           pvm_uplong(&clock_sec, 1, 1 );
25.           pvm_uplong(&clock_usec, 1, 1 );
26.           printf("Master: komunikat wyslany w chwili %ld:%ld \
                 odebrany w chwili %ld:%ld \n", clock_sec,    \
                 clock_usec, tv.tv_sec, tv.tv_usec );
27.     }
28.     pvm_exit();
28. }
```

Jak widzimy, pierwsza część programu jest praktycznie identyczna jak w poprzednich ćwiczeniach. Po utworzeniu (w linijce 13) procesów potomnych, master oczekuje na nadejście od nich komunikatów. W pętli z linijek 19-26 wykonuje funkcję blokującego odbierania, po odebraniu wiadomości pobiera wartość lokalnego zegara (linijka 23), wypakowuje wartość etykiety czasowej odebranej wiadomości (linijka 24-25) oraz wypisuje obie te wartości na ekranie (linijka 26).

Wreszcie, w linijce 28 proces wychodzi z środowiska PVM (jeszcze raz przypomnijmy, że funkcja `pvm_exit` nie kończy działania procesu!) po czym funkcja `main` się kończy a razem z nią i działanie procesu.

Funkcja `gettimeofday` służy w systemie Linuks do pobierania bieżącego czasu systemu. Wypełnia ona strukturę, której adres podany jest jako pierwszy parametr, liczbą sekund (pole `tv_sec`) i mikrosekund (pole `tv_usec`), które upłynęły od początku pierwszego stycznia 1970 roku (czasu uruchomienia pierwszego Uniksa). Zapoznać się możesz z nią za pomocą polecenia systemu `man gettimeofday`. Drugi parametr nie ma w Linuksie praktycznie zastosowania i jego stosowanie silnie się odradza, dlatego też tutaj jego wartość ustawiona jest na NULL.

Teraz napiszemy program przeznaczony dla procesów typu *slave*. Przypomnijmy, że ich zadaniem będzie wysłanie komunikatu, który będzie zawierał etykietę czasową określającą czas wysłania wiadomości.

`program slave.c`

```
1.      #include <sys/time.h>
2.      #include "def.h"

3.      int main()
4.      {
5.            int i,j;
6.            struct timeval tv;
7.
8.            pvm_initsend( PvmDataDefault);
9.            gettimeofday(&tv,NULL);
10.           pvm_pklong( &tv.tv_sec, 1, 1);
11.           pvm_pklong( &tv.tv_usec, 1, 1);
12.           pvm_send(pvm_parent(), MSG_SLV);
13.           pvm_exit();
14.     }
```

Tak samo jak w programie `master.c` musimy załączyć plik nagłówkowy `sys/time.h` zawierający potrzebne definicje struktury `timeval` oraz funkcji `gettimeofday`. Następnie musimy zdefiniować zmienną `tv` typu strukturalnego `timeval`, która będzie zawierała wartość lokalnego zegara fizycznego oraz która będzie etykietą czasową wiadomości wysyłanej do procesu macierzystego.

Proces *slave* inicjuje bufor komunikacyjny (linijka 8), pobiera czas rzeczywisty (linijka 9). Czas ten to para składająca się z liczby sekund i oraz mikrosekund, który następnie jest wstawiany jako etykieta czasowa do wysyłanego komunikatu (funkcja `pvm_pklong` w linijkach 10-11). Przygotowany komunikat wysyłany jest do procesu nadrzędnego (*mastera*, funkcja `pvm_send` w linijce 12), którego identyfikator pobierany jest za pomocą funkcji `pvm_parent`. Wreszcie proces opuszcza środowisko PVM i kończy swoje działanie.

Oba programy kompilujemy w zwykły sposób.

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

Po skompilowaniu wynikowe pliki wykonywalne umieszczamy w odpowiednim katalogu.

```
cp master slave $PVM_HOME
```

Następnie uruchom konsolę PVM i upewnij się, że maszyna wirtualna składa się co najmniej z kilku węzłów (polecenie konsoli `conf`). Teraz już możesz uruchomić przygotowany program za pomocą polecenia konsoli PVM `spawn -> master`. Za każdym razem wynik będzie oczywiście nieco inny i mogą się zdarzyć sytuacje, w których akurat, przypadkowo, nie da się zaobserwować rozbieżności czasowych. Wynik uruchomienia może wyglądać na przykład tak:

```
pvm> spawn -> master
[1]
1 successful
t80001
pvm> [1:t40002] EOF
[1:t140001] EOF
[1:tc0001] EOF
[1:t80001] Master: Moj identyfikator to 524289
[1:t80001] Master: komunikat wyslany w chwili 1153728881:108777
odebrany w chwili 1153728886:741902
[1:t80001] Master: komunikat wyslany w chwili 1153728887:308292
odebrany w chwili 1153728886:750846
[1:t80001] Master: komunikat wyslany w chwili 1153728891:6884
odebrany w chwili 1153728886:761332
[1:t80001] Master: komunikat wyslany w chwili 1153728887:791852
odebrany w chwili 1153728886:776174
[1:t100001] EOF
[1:t80001] EOF
[1] finished
```

W podanym przykładzie procesy zostały uruchomione na komputerach w sieci lokalnej, na których zegary są synchronizowane za pomocą serwera synchronizacji czasu wykorzystującego protokół *ntpd*. Mimo tego widzimy, że zdarzyły się sytuacje, w których proces *master* odebrał wiadomość *z przyszłości*. Na przykład widać wiadomość zawierającą etykietę czasową `1153728887:791852`, podczas gdy została odebrana w chwili `1153728886:776174`.

W ten sposób przekonaliśmy się, że używanie czasu rzeczywistego, zegarów fizycznych nie spełnia warunku koniecznego do poprawnego działania wielu algorytmów rozproszonych: zdarzenie odebrania wiadomości może poprzedzać zdarzenie wysłania wiadomości! Dodatkowo może się zdarzyć, że pewne wiadomości były wysyłane *natychmiastowo* (czas wysłania równy był czasowi odebrania wiadomości). Zakłóca to drugi warunek podany przez nas wcześniej, unikalności etykiet czasowych, ale ten problem można bardzo łatwo rozwiązać, traktując każdą wartość zegara jako parę dwóch liczb: identyfikatora zadania oraz właściwej wartości zegara fizycznego.

Zadanie do samodzielnego wykonania

Na wykładzie poznałeś algorytmy implementujące czas logiczny, który może być niezależny od zegara fizycznego. Twoim zadaniem obecnie będzie takie zmodyfikowanie naszego programu, by zamiast czasu rzeczywistego, fizycznego, używał właśnie czasu wirtualnego. Głównym celem jest zrozumienie budowy wybranego algorytmu oraz nabycie umiejętności przekładania wiadomości teoretycznych na praktyczne implementacje. Wybierzemy algorytm Lamporta realizujący ideę skalarnych zegarów logicznych, jako prosty i łatwy do zrozumienia i implementacji.

W programie `master.c` usuń linijki 1 (załączenie pliku nagłówkowego `sys/time.h`) oraz linijkę 10 (definicja zmiennej `tv`). Linijki te nie będą potrzebne, gdyż nie będziesz używać czasu rzeczywistego. Linijki 23-24 powinieneś zastąpić własnym kodem. W programie `slave.c` usuń linijkę 1 i 7. Zastąp własnym kodem (jeżeli będzie on potrzebny) linijki 10 oraz 11. Widzisz więc, że niezbędne zmiany nie są duże i powinieneś móc sobie łatwo poradzić z tym zadaniem.

Załóżmy dodatkowo, że początkową wartością zegarów logicznych jest jakaś wartość różna dla każdego zadania. Niech tą unikalną wartością będzie identyfikator PVM każdego zadania (gdyż, jak pamiętamy, w tym wypadku samo środowisko PVM dba o to, by identyfikator zadania był unikalny w obrębie całej maszyny wirtualnej). Dopiszemy więc jeszcze na samym początku linijkę:

```
local_clock=pvm_mytid();
```

Jak powinieneś pamiętać z wykładu, czas logiczny (implementowany przez algorytm Lamporta) charakteryzuje się następującymi własnościami:

- Każde zdarzenie lokalne powoduje zwiększenie zegara logicznego o pewną wartość *d* (na przykład może wynosić 1). U nas, wystarczy jak skupimy się na zdarzeniach odebrania i wysłania wiadomości
- Przy odebraniu wiadomości, a przed wykonaniem jakiejkolwiek innej operacji, uaktualnia się zegar logiczny tak, by zawsze był o co najmniej *d* większy od otrzymanej etykiety czasowej - tak więc czas przesyłania wiadomości powinien trwać co najmniej *d* jednostek czasu. W szczególności gdy *d* równa się jeden - jedną jednostkę czasu (logicznego).

Jak pamiętamy, unikalność etykiet czasowych wszystkich zdarzeń może być rozwiązywana poprzez traktowanie każdej wartości zegara jako pary składającej się z identyfikatora zadania i liczby określającej logiczny czas lokalny. Ponieważ nie jest to ważne w tym akurat ćwiczeniu, możesz tą kwestię pominąć (ale powinieneś o niej pamiętać!).

Rozwiązanie zadania znajduje się w materiałach kursu. Nie zaglądaj do niego, dopóki nie ukończysz ćwiczenia samodzielnie. Wynik uruchomionego programu (dla 10 procesów typu *slave*) może wyglądać na przykład tak:

```
pvm> spawn -> master
[1]
1 successful
t180008
pvm> [1:t4000d] EOF
[1:t4000e] EOF
[1:t80008] EOF
[1:t80009] EOF
[1:t180009] EOF
[1:t100009] EOF
[1:t10000a] EOF
[1:t140008] EOF
[1:t140009] EOF
[1:t180008] Master: Moj identyfikator to 1572872
[1:t180008] Master: komunikat wyslany w chwili 268806 odebrany w chwili
268807
[1:t180008] Master: komunikat wyslany w chwili 1572873 odebrany w chwili
1572874
[1:t180008] Master: komunikat wyslany w chwili 262158 odebrany w chwili
1572875
[1:t180008] Master: komunikat wyslany w chwili 262157 odebrany w chwili
1572876
[1:t180008] Master: komunikat wyslany w chwili 273347 odebrany w chwili
1572877
[1:t180008] Master: komunikat wyslany w chwili 1048585 odebrany w chwili
1572878
[1:t180008] Master: komunikat wyslany w chwili 1048586 odebrany w chwili
1572879
[1:t180008] Master: komunikat wyslany w chwili 1310728 odebrany w chwili
1572880
[1:t180008] Master: komunikat wyslany w chwili 1310729 odebrany w chwili
1572881
[1:t180008] Master: komunikat wyslany w chwili 786441 odebrany w chwili
1572882
[1:tc0009] EOF
[1:t180008] EOF
[1] finished
```

Widzimy tutaj, że początkowa wartość zegara *mastera* wynosiła 1572872 (taka sama jak identyfikator zadania). Po otrzymaniu wiadomości zegar zwiększany jest o jeden do wartości 1572873; Ponieważ jednak wiadomość ta została wysłana właśnie w chwili 1572873 a zakładamy, że wiadomości nie mogą być przesyłane natychmiastowo, zegar przesuwa się o wartość o jeszcze jeden większą do

1572874. Od tej pory żadna z etykiet czasowych nadsyłanych wiadomości nie jest większa od lokalnego zegara logicznego, więc jest on zawsze inkrementowany po każdorazowym zdarzeniu odbioru wiadomości.

W celu uzyskania większego obycia z biblioteką PVM oraz lepszego zaznajomienia się z ideą czasów logicznych, możesz dodatkowo napisać jeszcze kilka programów: wykorzystujących na przykład dodatkowo zegar fizyczny maszyny lub też algorytm Fidge-Matterna realizujący ideę zegarów wektorowych.

Stworzenie topologii pierścienia

Kolejny program, który napiszemy, będzie miał na celu demonstrację możliwości połączenia procesów w pierścień. Odbędzie się to dzięki przesłaniu przez *mastera* do wszystkich procesów tablicy zawierającej identyfikatory wszystkich procesów, dzięki czemu będą one mogły wybrać swoich sąsiadów w pierścieniu. Po połączeniu procesów w pierścień *master* prześle dalej wiadomość zawierającą licznik odwiedzin. Licznik ten będzie inkrementowany przez każdy z procesów przed przesłaniem dalej wiadomości.

`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 res = 0;

7.         pvm_spawn("slave",0,0,".",SLAVENUM,tids);

8.         pvm_initsend( PvmDataDefault );
9.         pvm_pkint(tids,SLAVENUM,1);
10.        pvm_mcast(tids,SLAVENUM,MSG_MSTR);

11.        pvm_initsend( PvmDataRaw );
12.        pvm_pkint( &res, 1,1);
13.        pvm_send(tids[0], MSG_SLV);

14.        pvm_recv( -1, MSG_SLV );
15.        pvm_upkint( &res, 1, 1);
16.        printf("Master: Token przeszedl pierscien: \
                                t%d\n",res);
17.        pvm_exit();
18. }
```

Pierwsze siedem linijek nie powinno już wymagać wyjaśnień. Obejmują one załączenie odpowiednich plików nagłówkowych, deklarację zmiennych i uruchomienie procesów potomnych.

Rozesłanie do wszystkich procesów *slave* tablicy `tids` (która została wypełniona identyfikatorami procesów utworzonych przez funkcje `pvm_spawn` w linijce 7) następuje w linijkach 8-10. Po zakończeniu rozsyłania *master* wysyła do swojego następnika w pierścieniu prosty komunikat zawierający liczbę 0. Następnikiem tym jest proces, którego identyfikator znajduje się w pierwszym wpisie tablicy `tids`.

Następnie *master* wywołuje blokującą funkcję odbioru komunikatów `pvm_recv` (linijka 14), czekając aż przesłana przez niego wiadomość przejdzie przez cały pierścień. Po otrzymaniu oczekiwanej wiadomości, wypisuje odpowiedni komunikat na standardowym wyjściu, opuszcza środowisko PVM i kończy pracę.

Obecnie przejdziemy do omówienia kodu programu `slave.c`, który będzie wykonywany przez procesy typu *slave*. Podstawową kwestią jest sprawa w jaki sposób ustalić następnika. Można by było

to zrobić na poziomie kodu wykonywanego przez *mastera*, który następnie rozesłałby każdemu procesowi potomnemu jego następnika. My jednak wybierzemy inny sposób, dzięki któremu unikniemy tego dodatkowego komunikatu.

`Program slave.c`

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

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

9.        pvm_recv(-1,MSG_MSTR);
10.       pvm_upkint( tids, SLAVENUM, 1);
11.       for (i=0;i<SLAVENUM-1;i++)
12.       {
13.             if ( tids[i] == tid )
14.             {
15.                   succ = tids[i+1];
16.                   break;
17.             }
18.       }
19.       if (!succ) succ = pvm_parent();

20.       printf("Slave: nastepnik w pierscieniu: t%x\n", \
                                                    succ);
21.       pvm_recv( -1, MSG_SLV );
22.       pvm_upkint( &res, 1, 1);
23.       res++;
24.       pvm_initsend( PvmDataRaw );
25.       pvm_pkint( &res, 1, 1);
26.       pvm_send( succ, MSG_SLV );
27.       pvm_exit();
28. }
```

Proces *slave* odbiera od *mastera* tablicę zawierającą identyfikatory wszystkich procesów *slave* (włącznie z nim samym) za pomocą funkcji `pvm_recv` (linijka 9). Wybór następnika następuje w linijkach 11-19. Proces wyszukuje swój własny identyfikator w tablicy i jako następnika wybiera identyfikator z kolejnego wpisu. Jeżeli proces był ostatnim w tablicy, jako następnika wybiera proces macierzysty (*mastera*, linijka 19).

Następnie proces *slave* odbiera od poprzednika wiadomość, wypakowuje z niej licznik odwiedzin (linijka 22), inkrementuje go (linijka 23) i przesyła do następnika (linijki 24-26). Wreszcie opuszcza środowisko PVM i kończy działanie.

Oba programy kompilujemy w zwykły sposób.

```
gcc master.c –o master –lpvm3

gcc slave.c –o slave –lpvm3
```

Po skompilowaniu wynikowe pliki wykonywalne umieszczamy w odpowiednim katalogu.

```
cp master slave $PVM_HOME
```

Przykładowy efekt uruchomienia programu za pomocą komendy konsoli PVM `spawn -> master` dla 4 procesów typu *slave* może wyglądać tak:

```
pvm> spawn -> master
[1]
1 successful
t100039
pvm> [1:t40057] Slave: Czekam na wiadomosci
[1:t40057] Slave: nastepnik w pierscieniu: t8003a
[1:t40057] EOF
[1:t140038] Slave: Czekam na wiadomosci
[1:t140038] Slave: nastepnik w pierscieniu: t40057
[1:t140038] EOF
[1:t8003a] Slave: Czekam na wiadomosci
[1:t8003a] Slave: nastepnik w pierscieniu: tc003a
[1:t100039] Master: Token przeszedl pierscien: t4
[1:tc003a] Slave: Czekam na wiadomosci
[1:tc003a] Slave: nastepnik w pierscieniu: t100039
[1:t8003a] EOF
[1:t100039] EOF
[1:tc003a] EOF
[1] finished
```

Podsumowanie

W wyniku tego ćwiczenia powinieneś zrozumieć, dlaczego używanie zegarów fizycznych w środowisku rozproszonym nie jest najlepszym rozwiązaniem, oraz stworzyłeś aplikację używającej algorytmu Lamporta do zaimplementowania idei zegarów logicznych.

Co powinieneś wiedzieć:

- Jakie zagrożenia niesie używanie zegarów fizycznych w rozproszonych aplikacjach
- Jaki jest cel oraz zasada działania algorytmy Lamporta