Instytut Teleinformatyki

Wydział Inżynierii Elektrycznej i Komputerowej
Politechnika Krakowska

programowanie usług sieciowych

„Wątki”

laboratorium: 07

Kraków, 2014

Spis treści

1. Wiadomości wstępne

Pierwsza część niniejszej instrukcji zawiera podstawowe wiadomości teoretyczne dotyczące **wątków** (ang. *Threads)* . Poznanie tych wiadomości umożliwi prawidłowe zrealizowanie praktycznej części laboratorium.

1.1. Tematyka laboratorium

Procesy w Uniksie mogą współpracować ze sobą. Mogą przesyłać do siebie komunikaty i przerywać sobie nawzajem pracę. Mogą nawet ustalić, że będą współdzielić segmenty pamięci. Jednak zasadniczo są odrębnymi bytami wewnątrz systemu operacyjnego. Niezbyt chętnie wymieniają się swoimi zmiennymi. Proces potomny tworzymy za pomocą funkcji `fork()`. Niestety mechanizm ten nie jest pozbawiony pewnych wad:

- Wywołanie funkcji `fork` jest kosztowne. Wymaga bowiem skopiowania zawartości obszaru pamięci przydzielonej procesowi macierzystemu do pamięci przydzielonej procesowi potomnemu, powielania w procesie potomnym wszystkich deskryptorów itd.,
- Przekazywanie informacji od procesu potomnego do procesu macierzystego jest bardzo uciążliwe.

Na szczęście istnieje klasa procesów znanych pod nazwą **wątków**. Różnią się one od procesów tym, że są odrębnymi strumieniami wykonawczymi działającymi w ramach pojedynczego procesu. Oznacza to, że program może współbieżnie wykonywać kilka swoich części, z uwzględnieniem ograniczeń sprzętowych danego systemu komputerowego. Na przykład edytor tekstu może dzielić dokument na strony, jednocześnie sprawdzając pisownię. Dwa lub więcej wątków działa wewnątrz tego samego procesu, pracując na tym samym zbiorze zmiennych i danych.
Chociaż wątki są dość trudne do programowania, mogą być bardzo wartościowym mechanizmem w niektórych aplikacjach, na przykład w wielowątkowych serwerach baz danych. Główną zaletą wątków jest to, że są procesami „wagi lekkiej" (ang. *Light weight processes*) w przeciwieństwie do procesów „wagi ciężkiej" (ang. *havy weight processes*), na przykład wykonujących zadania systemowe. Procesy wagi lekkiej nie obciążają kosztami związanymi z inicjacją przestrzeni adresowej i uruchomieniem procesu, jak czynią to zadania systemu operacyjnego. Jest to szczególnie opłacalne w przypadku krótkotrwałych procesów, ale odbywa się kosztem niezawodności: każdy wątek

(proces wagi lekkiej) może spowodować awarię wszystkich pozostałych. Utworzenie wątku trwa od dziesięciu do stu razy krócej niż utworzenie procesu.

1.2. Zagadnienia do przygotowania

Przed przystąpieniem do realizacji laboratorium należy zapoznać się z zagadnieniami dotyczącymi **tworzenia oraz obsługi wątków w systemie linux**:

- Różnice pomiędzy procesami, a wątkami i przy pomocy jakich mechanizmów (funkcji) je tworzymy;
- Sposoby synchronizacji procesów (mutexy, semafory, zmienne warunkowe);
- Funkcje bezpieczne dla wątków;
- Podstawowe operacje na wątkach: tworzenie i kończenie itp.
- Komunikacja międzywątkowa;
- Przekazywanie parametrów do wątków

Literatura:

[1] W.R. Stevens, *„Programowanie Usług Sieciowych”*, *„Wątki”*.
[2] N. Mathew, R. Stones, *„LINUX Programowanie”.*
[3] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

1.3. Cel laboratorium

Celem laboratorium jest zapoznanie się z techniką obsługi i tworzenia wątków. Wątki są bardzo pomocnym mechanizmem w tworzeniu aplikacji. Udostępniają wiele ciekawych funkcji oraz dają duże możliwości w tworzeniu wydajnych aplikacji. Podczas realizacji tego laboratorium zapoznasz się z:

- z zaletami stosowania wątków
- funkcjami tworzenia, kończenia wątków
- funkcjami synchronizacji wątków
- z technikami obsługi wątków.
- z zaletami wątków w odniesieniu do procesów.

2. Przebieg laboratorium

Druga część instrukcji zawiera zadania do praktycznej realizacji, które demonstrują zastosowanie technik z omawianego zagadnienia.

2.1. Przygotowanie laboratorium

W celu wykonania laboratorium należy pobrać przygotowane programy, które dostępne są w katalogu /home/shared/pus/pus_07_threads/ na serwerze mars.iti.pk.edu.pl. Katalog, w którym wykonywane będą kolejne ćwiczenia należy nazwać:

```
~/pus/pus_07_threads/
```

Każdy z programów ma przygotowany plik `Makefile`, przy pomocy którego programy będą kompilowane. Do wykonania ćwiczenia wymagany jest tylko kompilator `gcc` pod systemem linux. Jeśli ktoś zdecyduje się na samodzielną kompilację bez używania polecenia make, należy wymusić dołączenie biblioteki `<pthread.h>`, przez użycie opcji `-lpthread`.

```
cc -lpthread watki.c
```

W obserwacji działania wątków pomocna może być program top:

$ top

Uruchamiamy go w drugiej konsoli, opcje podczas działania programu:

- ***H – pokaż wszytskie wątki***
- ***u – wyświetlanie wątków tylko danego usera***
- ***h – help***

Wszystkie opcje i dokładny opis

$ man top

2.2. Zadanie 1. Tworzenie wątków

Zadanie pierwsze będzie polegało na przeanalizowaniu oraz uruchomieniu poniższego programu. Jest to program, który tworzy 2 wątki. Pokazuje on, iż wątki te dzielą wspólne zmienne.

Aby uruchomić program:

1. Przejdź do katalogu ze źródłami programu:
   ```
   $ cd ~/pus/pus_07_threads/zad1
   ```

2. Skompiluj źródła do postaci binarnej:
   ```
   $ make
   ```
   lub
   ```
   $ cc –lpthread –o watki watki.c
   ```

3. Uruchom program:
   ```
   $ ./watki
   ```

4. Na konsoli powinny pojawić się następujące dane:

   ```
   Nacisnij [x] oraz Enter, aby zakonczyc program
   Jestem watkiem glownym, moj licznik wynosi: 0
   Jestem watkiem glownym, moj licznik wynosi: 1
   Jestem watkiem glownym, moj licznik wynosi: 2
   Jestem watkiem glownym, moj licznik wynosi: 3
   Jestem watkiem glownym, moj licznik wynosi: 4
   Jestem watkiem glownym, moj licznik wynosi: 5
   Jestem watkiem glownym, moj licznik wynosi: 6
   x

   Na pewno zakonczyc? t/n
   n
   Jestem watkiem glownym, moj licznik wynosi: 7
   Jestem watkiem glownym, moj licznik wynosi: 8
   Jestem watkiem glownym, moj licznik wynosi: 9
   Jestem watkiem glownym, moj licznik wynosi: 10
   x
   Na pewno zakonczyc? t/n
   t
   ```

5. Zaobserwuj jak wątki współdzielą zmienne.

W programie tym uruchomione są równolegle dwa wątki. Pierwszy liczy w górę pod warunkiem że zmienna **end** jest równa *false*. Drugi wczytuje znak wprowadzony z klawiatury i odpowiednio może zmienić zmienną **end** na *true* co spowoduje zakończenie pierwszego wątku. W efekcie nie musimy za każdym razem zatrzymywać programu i czekać na reakcję użytkownika. Program zareaguje dopiero wtedy gdy jakiś znak zostanie wprowadzony.

Zadanie dodatkowe – patrząc na kod programu *watki.c* znajdź, które zmienne są współdzielone przez obydwa wątki i znajdź potwierdzenie tego.

2.3. Zadanie 2. Pobieranie wyników z wątków

Zadanie drugie będzie polegało na przeanalizowaniu oraz uruchomieniu poniższego programu. Jest to program, który tworzy 2 wątki wykonujące równoległe obliczenia. Celem ćwiczenia będzie zaobserwowanie sposobu pobierania wyników z poszczególnych wątków do programu głównego.

Aby uruchomić program:

1.Przejdź do katalogu ze źródłami programu:

```
$ cd ~/pus/pus_07_threads/zad2
```

2.Skompiluj źródła do postaci binarnej:

```
$ make
lub
$ cc –o watki2 watki2.c
```

3.Uruchom program:

```
$ ./watki2
```

4.Na konsoli powinny pojawić się następujące dane:

```
Wartosc poczatkowa zmiennych wynosi 0
Watek 1 zwieksza wartosc o 4
Watek 2 zwieksza wartosc o 5

Wartość zwrócona przez wątek 1 40
Wartość zwrócona przez wątek 2 50

Dla porownania poprawnosci
Wyniki otrzymane ze zmiennych globalnych na ktorych operowaly watki
Zmienna wykorzystawana przez watek 1 ma wartosc 40
Zmienna wykorzystawana przez watek 2 ma wartosc 50
```


Nasz program operował na zmiennych globalnych aby było możliwe pokazanie poprawności pobieranych danych z wątków. Na dwóch zadeklarowanych zmiennych wykonywał odpowiednie funkcje a wynik przekazywał do głównej funkcji. Jak widzimy wyniki otrzymane są równe co jest dowodem poprawnego odczytania wartości z wątków.

Zadanie dodatkowe – należy przeglądnąć kod programu `watki2.c` i zmodyfikować stałe wartości występujące w programie . Po dokonanych zmianach sprawdzić poprawność przekazywania . Można również zmienić operacje występujące w funkcjach na bardziej złożone i również wykonać testy. Student może poeksperymentować z typami danych aby zobaczyć jak będzie się zachowywał program.

2.4. Zadanie 3. Dane prywatne wątku

Zadanie trzecie będzie polegało na pokazaniu w jaki sposób stworzyć i używać prywatnych danych każdego z wątków. W przeciwieństwie do procesów, wszystkie wątki w pojedynczym programie dzielą tą samą przestrzeń adresową. Oznacza to, że modyfikacja np. zmiennej globalnej jest od razu zauważana w pozostałych wątkach. Umożliwia to znacznie łatwiejszą komunikację niż w przypadków procesów.

Jednakże pomimo współdzielenia tej samej przestrzeni adresowej każdy z wątków ma swój własny stos. Dzięki temu wątek może wykonywać osobny fragment kodu i wywoływać i powracać z funkcji w "zwykły" sposób.

Czasami jednak chcielibyśmy zrezygnować z współdzielenia pewnego zasobu globalnego dla wszystkich wątków. Chcemy aby każdy wątek miał swoją własną "prywatną" zmienną. Aby to lepiej opisać warto skorzystać z klasycznego problemu:

Mamy program, który monitoruje pewne zasoby/sprzęt. Program ten składa się z kilku wątków - każdy z nich jest odpowiedzialny za monitorowanie innej części zasobu i zapisuje swój stan w osobnym pliku logu. I właśnie chcemy sprawić aby te pliki logu były prywatnym polem wątku. Pierwszą czynnością do zrobienia jest utworzenie klucza reprezentującego zasób.

Tworzymy go dzięki następującej funkcji:

```
#include <pthread.h>

pthread_key_create(pthread_key_t * klucz,
                                  void(* funkcja_czyszczaca, void*));
```

Jej działanie zostanie wyjaśnione w poniższym przykładzie. Drugim argumentem jest wskaźnik do funkcji, która zostanie wywołana w chwili zakończenia wątku.

Następnie w każdym wątku kojarzymy zmienną którą chcemy traktować jako prywatną z utworzonym kluczem.

Używamy do tego celu funkcji:

```
#include <pthread.h>

pthread_set_specific(pthread_key_t klucz, void * dane_prywatne ));
```

Do odczytu danych prywatnych możemy posługiwać się funkcją:

```
#include <pthread.h>

pthread_get_specific(pthread_key_t klucz, void * dane_prywatne ));
```

Poniżej przedstawiona jest przykładowa implementacja tego problemu wraz z opisem działania.

```
/* Importujemy potrzebne biblioteki */
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>

/* Klucz uzywany do powiazania kazdego pliku logu z watkiem */
static pthread_key_t thread_log_key;

/* Funckja zapisujaca jakis komunikat do pliku */

void write_to_thread_log (const char* message)
{
   FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
   fprintf (thread_log, "%s", message);
}

/* Funkcja "czyszczaca" */

void close_thread_log (void* thread_log)
{
   fclose ((FILE*) thread_log);
}

void* thread_function (void* args)
{
   char thread_log_filename[20];
   FILE* thread_log;
```



```
   /* Tworzymy nazwe pliku dla kazdego logu */
   sprintf (thread_log_filename, "thread%d.log", (int) pthread_self
());
   /* Otwieramy plik */
   printf("Watek tworzy plik logu thread%d.log\n",
(int)pthread_self());
   thread_log = fopen (thread_log_filename, "w");
   /* Skladujemy wskaznik do pliku w sekcji prywatnej watku */
   pthread_setspecific (thread_log_key, thread_log);

   write_to_thread_log ("Watek monitorujacy startuje. \n");
   /* Tutaj wykonujemy jakies czynnosci */

   write_to_thread_log ("Watek monitorujacy konczy dzialanie.\n");
   return NULL;
}

int main ()
{
   int i;
   pthread_t threads[5];

   /* Tworzymy klucz sluzacy do skojarzenia kazdego pliku logu z
odpowiadajacym mu watkiem */
   pthread_key_create (&thread_log_key, close_thread_log);
   /* Tworzymy watki */
   for (i = 0; i < 5; ++i)
     pthread_create (&(threads[i]), NULL, thread_function, NULL);
   /* Czekamy az watki skoncza */
   for (i = 0; i < 5; ++i)
     pthread_join (threads[i], NULL);
   return 0;
}
```

Aby zaobserwować działanie programu:

1.Przejdź do katalogu ze źródłami programu:

```
$ cd ~/pus/pus_07_threads/zad3
```

2.Skompiluj źródła do postaci binarnej:

```
$ make
lub
$ cc –o watki3 watki3.c
```


3.Uruchom program:

```
$ ./watki3
```

4.Na konsoli powinny pojawić się następujące dane:

```
Watek tworzy plik logu thread1100364720.log
Watek tworzy plik logu thread1083587504.log
Watek tworzy plik logu thread1091976112.log
Watek tworzy plik logu thread1108757424.log
Watek tworzy plik logu thread1117146032.log
```

6. Przeanalizuj działanie programu, przejrzyj logi oraz wyciągnij wnioski.

2.5. Zadanie 4. Przekazywanie argumentów do wątku

Zadanie polega na napisaniu prostego programu obrazującego przekazywanie argumentów do wątku.

Są dwa sposoby przekazywania argumentów do wątku:

- przekazywanie wszystkich niezbędnych informacji przy pomocy wskaźnika do struktury zrzutowanej na typ void* (wynika to z wymaganej postaci „funkcji startowej”, której jedyny parametr wywołania jest właśnie typu void*),
- przekazanie jedynie indeksu do tablicy globalnej, gdzie trzymane są informacje o wszystkich parametrach procesów.

W tym zadaniu wykorzystany zostanie pierwszy sposób.

Na początku należy pobrać szablon drugiego programu (jego położenie jest opisane w p. 2.1.).

Należy w odpowiednim miejscu utworzyć wątki jednocześnie przekazując im zdefiniowane wcześniej argumenty za pomocą funkcji:

```
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t * attr, \
void * (*start_routine)(void *), void * arg);
```

Tej samej funkcji używaliśmy w poprzednim zadaniu, jednak w miejsca argumentu `void * arg` wstawiona była tam wartość NULL. Tym razem użyjemy go do przekazania argumentów do wątku. Należy pamiętać o zrzutowaniu struktury argumentów wąt-

ku (`struct watki_dane`) na typ `void*`. W szablonie wskazane jest miejsca, gdzie należy wstawić funkcję.

W funkcji `watekStart( void * argumenty )` znajduje się kod wykonywany przez utworzone wątki. Należy dopisac do niej kod wypisujący argumenty przekazane do wątku. Należy pamiętać o zrzutowaniu argumentów na strukturę wcześniej zdefiniowana w programie (`struct watki_dane`).

Program kompilujemy poleceniem:

```
$ gcc watki4.c –lpthread –o watki4
```

A uruchamiamy:

```
$ ./watki4
```

Wyświetlą się informacje o argumentach przekazanych do wątków. Należy sprawdzić czy są one takie same jak zdefiniowane w kodzie programu.

2.6. Zadanie 5. Pobieranie wartości zwracanej przez wątek

Należy przetestować działanie gotowego programu pokazującego działanie funkcji `pthread_join()`. Program ten znajduje się w lokalizacji podanej w p. 2.1.

Po utworzeniu wątku i przypisaniu mu funkcji, stosujemy:

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr)
```

Parametrami są wątek oraz wskaźnik do miejsca w pamięci w które zwrócona zostanie wartość wykonanej przez podany wątek funkcji.

Zadaniem funkcji `funkcja_watku( void * parametr )` jest po prostu zwrócenie wartości typu `int`.

Kompilujemy program:

```
$ gcc watki5.c -lpthread –o watki5
```


Uruchamiamy:

```
$ ./watki5
```

Program zwróci wartość wyliczoną przez funkcję `funkcja_watku() np.:`

```
Wartość zwrócona przez wątek: 500
```

2.7. Zadanie 6. Wydajność obliczeń na wątkach *

Należy napisać dwie implementacje programu liczącego iloczyn macierzy przez wektor : pierwsza, czysto sekwencyjna i druga kwazirównoległa (czy też psełdorównoległa) na wątkach. Celem ćwiczenia jest sprawdzenie wydajności w/w programów w przypadku, gdy liczony problem (macierz i wektor) mieści się w fizycznej pamięci operacyjnej oraz kiedy ilość danych potrzebnych do zapamiętania macierzy i wektora jest dużo większa niż fizyczna pamięć ram komputera.
Dla ułatwienia macierz i wektor wylosować na wejściu programu (wyniki nas w tym przypadku nie interesują – program ma wyświetlać jedynie czas jaki komputer spędził na rozwiązywaniu problemu. Ponadto przyjąć, że macierz jest kwadratowa. Programy mają przyjąć jako parametr wielkość macierzy.
Wskazówki i wymagania:
Program wątkowy ma „emulować” system rozproszony – każdy wątek ma być wirtualnym procesorem liczącym iloczyn skalarny wektora przez wskazaną kolumnę macierzy. Program główny jest nadzorcą rozdzielającym zadania poszczególnym wątkom – w momencie kiedy są jeszcze pozostałe do przemnożenia kolumny macierzy należy je kolejno przydzielić wolnym wątkom, lub tym które zgłosiły, że skończyły przetwarzanie poprzedniego zadania. Program wątkowy ma przyjmować jako drugi argument ilość uruchamianych wątków.
Wskazówka pomiaru czasu:
Do pomiaru czasu użyć funkcji gettimeofday(), która daje informacje o czasie z rozdzielczością 1 us. Należy dołączyć plik nagłowkowy:
#include <sys/time.h>
Wnioski z zanotowanych wyników należy skomentować.

3. Opracowanie i sprawozdanie

Realizacja laboratorium pt. *„Wątki”* polega na wykonaniu wszystkich zadań programistycznych podanych w drugiej części tej instrukcji. Wynikiem wykonania powinno być sprawozdanie w formie wydruku papierowego dostarczonego na kolejne zajęcia licząc od daty laboratorium, kiedy zadania zostały zadane.
Sprawozdanie powinno zawierać:

- opis metodyki realizacji zadań (system operacyjny, język programowania, biblioteki, itp.),
- algorytmy wykorzystane w zadaniach (zwłaszcza, jeśli zastosowane zostały rozwiązania nietypowe),
- opisy napisanych programów wraz z opcjami,
- trudniejsze kawałki kodu, opisane tak, jak w niniejszej instrukcji,
- uwagi oceniające ćwiczenie: trudne/łatwe, nie/realizowalne podczas zajęć, nie/wymagające wcześniejszej znajomości zagadnień (wymienić jakich),
- wskazówki dotyczące ewentualnej poprawy instrukcji celem lepszego zrozumienia sensu oraz treści zadań.

