Programowanie współbieżne (Info II) 19/20

Zadanie zaliczeniowe 2: Asynchroniczne C

Wprowadzenie

Pula wątków to mechanizm pozwalający na uzyskanie współbieżnego wykonywanie wielu zadań w ramach jednego programu. W skład puli wchodzi pewna liczba wątków roboczych (ang. *worker threads*), czekających na pojawienie się pracy do wykonania.

Użycie puli wątków pozwala uniknąć ciągłego powoływania i czekania na zakończenie się wątku przy wykonywaniu krótkotrwałych zadań współbieżnych. Pozwala też na wykonywanie bardzo dużej liczby zadań niezależnie od siebie w sytuacji, gdy liczba dostępnych potoków przetwarzania jest ograniczona.

Polecenie

- Zaimplementuj pulę wątków zgodnie z poniższym opisem szczegółowym (3 pkt).
- Zaimplementuj obliczenia *future* zgodnie z poniższym opisem szczegółowym (3 pkt).
- Napisz program przykładowy **macierz**, obliczający za pomocą puli wątków sumy wierszy z zadanej tablicy (1 pkt).
- Napisz program przykładowy **silnia**, obliczający za pomocą mechanizmu *future* silnię zadanej liczby (1 pkt).
- Zadbaj, aby kod był napisany w sposób klarowny i rzetelny zgodnie z poniższymi wytycznymi. (2 pkt).

Szczegółowy opis puli wątków

Pulę wątków należy zaimplementować jako realizację interfejsu przedstawionego w pliku `"threadpool.h"`. Zamieszczone tam są m.in. następujące deklaracje:

```
typedef struct runnable {
  void (*function)(void *, size_t);
  void *arg;
  size_t argsz;
} runnable_t;

typedef struct thread_pool {

} thread_pool_t;

int thread_pool_init(thread_pool_t *pool, size_t pool_size);

void thread_pool_destroy(thread_pool_t *pool);

int defer(thread_pool_t *pool, runnable_t runnable);
```

Wywołanie `thread_pool_init` inicjuje argument wskazywany przez `pool` jako nową pulę, w której będzie funkcjonować `pool_size` wątków obsługujących zgłoszone do wykonania zadania. Za gospodarkę pamięcią wskazywaną przez `pool` odpowiada użytkownik biblioteki. Poprawność działania biblioteki jest gwarantowana tylko, jeśli każda pula stworzona przez `thread_pool_init` jest niszczona przez wywołanie `thread_pool_destroy` z argumentem reprezentującym tę pulę.

Wywołanie `defer(pool, runnable)` zleca puli wątków `pool` wykonanie zadania opisanego przez argument `runnable`, argumenty `function` są przekazywane przez wskaźnik `args`, w polu `argsz` znajduje się długość dostępnego do pisania i czytania buforu znajdującego się pod tym wskaźnikiem. Za zarządzanie pamięcią wskazywaną przez `args` odpowiada klient biblioteki.

Funkcja `function` powinna zostać obliczona przez wątek z puli `pool`; wywołanie `defer` może zablokować wywołujący je wątek, ale jedynie na potrzeby rejestracji zlecenia: powrót z `defer` jest niezależny od powrotu z wykonania `function` przez pulę.

Zadania zlecone do wykonania przez `defer` powinny móc wykonywać się współbieżnie i na tyle niezależnie od siebie, na ile to możliwe. Można ograniczyć liczbę współbieżnie wykonywanych zadań do rozmiaru puli. Pula w czasie swojego działania nie powinna powoływać więcej wątków niż określono parametrem `pool_size`. Utworzone wątki są utrzymywane aż do wywołania `thread_pool_destroy`.

Zastanów się nad tym, jak zrealizować powyższą bibliotekę tak, aby wykonywała się możliwie sprawnie na współczesnych komputerach. Postaraj się zrealizować taką implementację.

Szczegółowy opis mechanizmu obliczeń *future*

Przy pomocy puli wątków i operacji `defer` należy zaimplenentować asynchroniczne obliczenia `future` jako realizację interfejsu przedstawionego w pliku `"future.h"`. Zamieszczone są tam m.in. następujące deklaracje:

```
typedef struct callable {
  void *(*function)(void *, size_t, size_t *);
  void *arg;
  size_t argsz;
} callable_t;

typedef struct future {
} future_t;

int async(thread_pool_t *pool, future_t *future, callable_t callable);

int map(thread_pool_t *pool, future_t *future, future_t *from,
        void *(*function)(void *, size_t, size_t *));

void *await(future_t *future);
```

Wywołanie `int err = async(pool, future_value, callable)` inicjuje pamięć wskazywaną przez `future_value`. Za zarządanie tą pamięcią odpowiada użytkownik biblioteki. Na puli `pool` zlecane jest wykonanie `function` z argumentu `callable`. Funkcja `function` zwraca wskaźnik do wyniku. Użytkownik biblioteki powinien zadbać, żeby poprawnie ustawiła też rozmiar wyniku wykorzystując do tego celu trzeci argument typu `size_t*`.

Wołający może teraz:

- Zaczekać na zakończenie wykonania funkcji `function` przez wywołanie:

  ```
  void *result = await(future_value);
  ```

  Za gospodarkę pamięcią wskazywaną przez wskaźnik `result` odpowiada użytkownik biblioteki (pamięć ta może zostać przekazana do funkcji `function` za pomocą jej argumentów lub w tej funkcji zaalokowana).

- Zlecić jakiejś puli, niekoniecznie tej, która zainicjowała `future_value`, wywołanie innej funkcji na wyniku:

  ```
  err = map(pool2, mapped_value, future_value, function2);
  ```

Programy, w których aktywnie działa jakaś pula wątków, powinny mieć automatycznie ustawioną obsługę sygnałów. Ta obsługa powinna zapewniać, że program po otrzymaniu sygnału (SIGINT) zablokuje możliwość dodawania nowych zadań do działających pul, dokończy wszystkie obliczenia zlecone dotąd działającym pulom, a następnie zniszczy działające pule.

Dla ułatwienia implementacji można założyć, że zaimplementowana biblioteka będzie testowana w taki sposób, iż wątki nie będą ginęły w testach.

Opis programu macierz

Program **macierz** ma ze standardowego wejścia wczytywać dwie liczby k oraz n, każda w osobnym wierszu. Liczby te oznaczają odpowiednio liczbę wierszy oraz kolumn macierzy. Następnie program ma wczytać k*n linijek z danymi, z których każda zawiera dwie, oddzielone spacją liczby: v, t. Liczba v

umieszczona w linijce i (numerację linijek zaczynamy od 0) określa wartość macierzy z wiersza floor(i/n) (numerację kolumn i wierszy zaczynamy od 0) oraz kolumny i mod n. Liczba t to liczba milisekund, jakie są potrzebne do obliczenia wartości v. Oto przykładowe poprawne dane wejściowe:

```
2
3
1 2
1 5
12 4
23 9
3 11
7 2
```

Takie dane wejściowe tworzą macierz od dwóch wierszach i trzech kolumnach:

```
|  1  1 12 |
| 23  3  7 |
```

Program ma za zadanie wczytać tak sformatowane wejście (można zakładać, że podawane będą tylko poprawne dane), a następnie za pomocą puli wątków zawierającej 4 wątki policzyć sumy wierszy, przy czym pojedyncze zadanie obliczeniowe powinno podawać w wyniku wartość pojedynczej komórki macierzy, odczekawszy liczbę milisekund, które zostały wczytane jako potrzebne do obliczenia tej wartości (np. zadanie obliczeniowe wyliczenia wartości 3 z macierzy powyżej powinno odczekiwać 11 milisekund). Po obliczeniu należy wypisać sumy kolejnych wierszy na standardowe wyjście, po jednej sumie w wierszu. Dla przykładowej macierzy powyżej umieszczonej w pliku `data1.dat` wywołanie:

```
$ cat data1.dat | ./macierz
```

powinno spowodować pojawienie się na wyjściu

```
14
33
```

Opis programu silnia

Program **silnia** powinien wczytywać ze standardowego wejścia pojedynczą liczbę n, a następnie obliczać za pomocą puli 3 wątków liczbę n!. Po obliczeniu tej liczby wynik powinien zostać wypisany na standardowe wyjście. Program powinien wyliczać silnię, wykorzystując funkcję `map` i przekazując jej w `future_value` częściowe iloczyny. Dla przykładu wywołanie:

```
$ echo 5 | ./silnia
```

powinno spowodować pojawienie się na wyjściu

```
120
```

Wymagania techniczne

Do synchronizacji można korzystać tylko z mechanizmów biblioteki `pthreads`. Można korzystać z plików nagłówkowych:

```
#include <pthread.h>
#include <semaphore.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

Powyższa lista może ulec rozszerzeniu, jeśli okaże się to konieczne. Można założyć, że kod będzie kompilowany i testowany na serwerze `students`.

Jako rozwiązanie należy wysłać na moodla plik `ab123456.tar.gz`, gdzie `ab123456` to login na students. W archiwum powinien znaleźć się jeden katalog o nazwie `ab123456` (login na students) z wszystkimi plikami rozwiązania. Kod programów przykładowych należy umieścić w plikach `macierz.c` i `silnia.c`. Nie powinna być konieczna modyfikacja plików `CMakeLists.txt`. Ciąg poleceń:

```
tar -xf ab123456.tar.gz
mkdir build && cd build
cmake ../ab123456
make
```

Powinien skompilować bibliotekę do pliku `build/libasyncc.a`, kompilator nie powinien wypisywać żadnych ostrzeżeń. Następnie można przetestować swoje rozwiązanie:

```
make test
```

W czasie oceniania zawartość katalogu `test` zostanie podmieniona.

Na pytania do zadania odpowiadają Piotr Cyrankiewicz i Aleksy Schubert.

Status przesłanego zadania

Status przesłanego zadania	Nie próbowano
Stan oceniania	Nie ocenione
Termin oddania	Tuesday, 7 January 2020, 20:00 PM
Pozostały czas	24 dni 2 godz.
Ostatnio modyfikowane	-
Komentarz do przesłanego zadania	▶ Komentarze (0)