Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium:
Wątki POSIX

Przedmiot: Przetwarzanie Równoległe i Rozproszone

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: dr inż. Mateusz Sitko

Data: 6 listopada 2024

Numery lekcji: 3, 4, 5

Grupa laboratoryjna: 4

Cel Zajęć

Celem zajęć było zapoznanie nas z wątkami POSIX w języku C. Przekazanie wiedzy na temat ich użycia, synchronizacji oraz przekazywania i dekompozycji danych.

Wątki POSIX

Użycie wątku

Wątki tworzymy wykorzystując funkcje **pthread_create:**

```
int pthread_create(
      pthread_t* thread,
      const pthread_attr_t* attr,
      void* (*start_routine)(void*),
      void* arg);
```

Która jako argumenty przyjmuje:

- `pthread_t*` – adres identyfikatora wątku.
- `pthread_attr_t*` – zestaw atrybutów.
- `void* (*start_routine)(void*)` – funkcja wykonywana przez wątek.
- `void*` – argument przekazywany funkcji.

Typem zwracany jest numer błędu `int` lub 0 w przypadku powodzenia.

Przykład użycia:

```
// Create a thread
pthread_t thread;
int rc = pthread_create(&thread, NULL, thread_func, NULL);
if (rc != 0)
      printf("Thread creation failed!\n");
```

W przypadku powodzenia wątek będzie wykonywał przekazaną formą wskaźnika funkcje:

```
void* thread_func(void* arg)
{
    printf("Thread is active!\n");
}
```

Zwrot wartości funkcji wątku

Aby zapobiec przedwczesnemu zakończeniu pracy programu przed pełnym wykonaniem się wątku oraz aby uzyskać zwróconą przez funkcje wątku wartość możemy wywołać na nim funkcje **pthread_join**.

```
int pthread_join(pthread_t thread, void** retval);
```

Która jako argumenty przyjmuje:

- `pthread_t` – identyfikator wątku.
- `void**` - adres w pamięci gdzie zostanie zapisana wartość zwrócona przez funkcje wykonywaną przez wątek.

Typem zwracany jest numer błędu `int` lub 0 w przypadku powodzenia.

Wątek, który wywołał funkcje `pthread_join` zostanie zatrzymany dopóki wątek, którego identyfikator został przekazany nie zakończy swojej pracy.

Aby funkcja wątku poprawnie zwróciła wartość za pośrednictwem funkcji `pthread_join`, wartość musi być zwrócona poprzez funkcje **pthread_exit**.

```
void pthread_exit(void* retval);
```

Która jako argument przyjmuje:

- `void*` - adres w pamięci, który zostanie przekazany `pthread_join`.

Przykład zastosowania:

```
void* thread_func(void* arg)
{
    int* return_code = malloc(sizeof(int));
    *return_code = 0;

    pthread_exit(return_code);
}
```

W powyższym przykładzie deklarujemy zmienną dynamiczną ponieważ wartości statyczne zostaną usunięte w momencie zakończenia pracy wątku.

Oczywiście oznacza to potrzebę zwolnienia tej pamięci w wątku nadrzędnym. Przykład z wykorzystaniem wielu wątków potomnych:

```
for (int i = 0; i < thread_number; i++)
{
     if (thread_created[i])
     {
          // Store thread's work in a buffer for easy casting
          void* buffer;
          pthread_join(threads[i], &buffer);

          // Cast the return to an integer
          done_work[i] = *(int*)(buffer);

          // Free the buffer
          free(buffer);
     }
     else
          done_work[i] = 0;
}
```

Alternatywnymi sposobami odzyskania wartości zwróconej przez wątek są:

- przekazanie adresu, gdzie wątek zapisze efekty swojej pracy, jako argumentu dla funkcji wątku.
- Użycie zmiennej globalnej do której funkcja wątku będzie mieć dostęp.

Przekazanie argumentu funkcji wątku

Dane do funkcji wątku możemy przekazać poprzez użycie czwartego argumentu funkcji.

```
int pthread_create(
     pthread_t* thread,
     const pthread_attr_t* attr,
     void* (*start_routine)(void*),
     void* arg);
```

Przykład użycia dla wielu wątków potomnych:

```
for (int i = 0; i < thread_number; i++)
{
      thread_ids[i] = i;
      int rc = pthread_create(&threads[i], NULL, thread_func, &thread_ids[i]);
      if (rc != 0)
            printf("ERROR: Failed to create thread %d. Reason: %d\n", i, rc);
}
```

Tablica **thread_ids** jest potrzebna, ponieważ bez niej każda iteracja pętli for nadpisywałaby wartość ID, przekazywaną do wątku. Gdybyśmy bezpośrednio przekazywali adres zmiennej **i**, wszystkie wątki odczytywałyby ten sam adres, który zmieniałby się w każdej iteracji, a dane mogłyby już nie być aktualne w momencie odczytu.

Użycie przez funkcje typu **void*** pozwala nam na przekazanie adresu do dowolnego typu danych w pamięci, niezależnie od tego czym może on być, jednak wymaga to od nas odpowiedniego potraktowania przekazanych danych poprzez rzutowanie.

```
void* thread_func(void* arg)
{
    // Cast the argument to an integer
    int thread_id = *(int*)arg;
    printf("Thread: %d is active\n", thread_id);

    pthread_exit(NULL);
}
```

Zdarzają się sytuacje kiedy chcemy przekazać wątkowi więcej informacji niż pojedynczą wartość. W języku C do tego możemy wykorzystać słowo kluczowe **struct**.

```
struct thread_args
{
    int id;
    double x1, y1;
    double x2, y2;
    double result;
};
```

Wykorzystanie `struct` pozawala nam na zdefiniowanie dowolnego zestawu danych jaki przekażemy wątkowi. Pośród tych danych możemy również zdefiniować zmienną, która przechowa wynik, `result` w przykładzie powyżej.

Elementy tablicy takich zestawów możemy wypełnić danymi zaraz przed utworzeniem wątku.

```
// Create threads
for(int i = 0; i < thread_number; i++)
{
     // Fill the structs with example values
     args[i].id = i;
     args[i].x1 = i / 16.0;
     args[i].x2 = i / 2.0;
     args[i].y1 = i / 4.0;
     args[i].y2 = i * 2.0;
     args[i].result = 0.0;   // Initialization for safety

     int rc = pthread_create(&threads[i], NULL, thread_func, &args[i]);
     if (rc != 0)
          printf("ERROR: Creation of thread %d failed! Reason: %d\n", i, rc);
}
```

Oczywiście, po przekazaniu musimy wykonać odpowiednie rzutowanie. Przykład użycia takiej struktury:

```
void* thread_func(void* args)
{
    // Cast given argument to the correct format
    struct thread_args *arg = (struct thread_args*)args;

    // Do an example computation
    double x = arg->x1 * arg->x2 * -1.0;
    double y = arg->y1 * arg->y2 * 0.1;

    arg->result = x + y;
    // arg->result = 1; // More predictable result
    printf("THREAD %d\tResult: %lf\n", arg->id, arg->result);

    // End the thread
    return 0;

}
```

Muteksy

Występują sytuacje, w których chcemy aby kilka wątków modyfikowało tą samą zmienną, np. poprzez zwiększanie sumy całkowitej składowe obliczone przez każdy wątek. Takie sytuacje mogą prowadzić do wystąpienia tak zwanego **Race Condition**.

Race condition - sytuacja wyścigu, występuje, gdy kilka wątków próbuje jednocześnie zmodyfikować tę samą zmienną bez odpowiedniej synchronizacji, co może skutkować nieprzewidywalnymi wynikami, gdyż kolejność wykonania operacji nie jest gwarantowana. Aby zarządzać dostępem do współdzielonych zasobów i uniknąć wyścigu, używamy muteksów**.**

Muteks - mechanizm synchronizacji, który zapewnia, że tylko jeden wątek może posiadać dostęp do określonego fragmentu kodu w danym momencie, co pozwala na bezpieczne i spójne modyfikacje zmiennych współdzielonych przez wiele wątków.

Muteks jest przechowywany w typie `pthread_mutex_t`. Jego inicjalizacja może nastąpić na dwa sposoby:

- `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;` - Inicjalizacja statyczna.
- `pthread_mutex_init(&mutex, NULL);` - Inicjalizacja dynamiczna.

Inicjalizacja dynamiczna jest używana, gdy potrzebujemy niestandardowych atrybutów muteksu lub chcemy zainicjalizować muteks w trakcie działania programu (np. w dynamicznie tworzonych obiektach). W jej przypadku muteks musi następnie zostać poprawnie wyczyszczony:

```
pthread_mutex_destroy(&mutex);
```

Z użyciem muteksów powiązane są trzy ważne metody:

- `pthread_mutex_lock(&mutex)` – Blokuje muteks, jeżeli muteks jest już zablokowany to egzekucja wątku zostanie wstrzymana do czasu jego odblokowania.
- `pthread_mutex_trylock(&mutex)`- Blokuje muteks I zwraca wartość 0 jeżeli muteks nie był jeszcze zablokowany. Przy użyciu tej funkcji wątek nie będzie czekać tylko będzie kontynuować swoją pracę
- `pthread_mutex_unlock(&mutex)` – odblokowuje muteks.

Przykład użycia:

```
void* thread_func(void* arg)
{
    // Lock the mutex before modifying global counter
    pthread_mutex_lock(&mutex);

    // Increment the global counter
    global_counter++;

    // Unlock the mutex after modification
    pthread_mutex_unlock(&mutex);

    return 0;
}
```

Zatrzymywanie wszystkich wątków nie będących w sekcji krytycznej może być czasochłonne i przesadnie spowalniać program, jednak jest to konieczne w przypadku pojedynczej zmiennej. W przypadku wielu zmiennych, np. tablicy możemy wykorzystać jednak funkcje `pthread_mutex_trylock`.

W poniższym przykładzie wątek próbuje uzyskać dostęp do pierwszego wolnego zasobu w tablicy. Jeżeli zasób będzie już zajęty przez inny wątek, jego zamknięcie się nie powiedzie i wątek spróbuje zarezerwować kolejny zasób – operacja powtarza się do skutku, aż uda się uzyskać dostęp do jednego z zasobów.

```
int reserve_a_tap()
{
    int i = 0;
    while (true)
    {
        if(pthread_mutex_trylock(&taps[i].mutex) == 0)
        {
            // Reserve the tap
            beer_taps[i].fills += 1;

            // Return the index of the tap
            return i;
        }
        else
        {
            // Change the index to try another tap
            i += 1;
            i %= TAP_NUMBER;
        }
    }
}
```

Niepoprawne użycie muteksów może prowadzić do poważnych problemów. Na przykład:

- **Zakleszczenie** - to sytuacja, w której dwa lub więcej wątków czeka na siebie nawzajem, trzymając zasoby, których potrzebują inne wątki do kontynuowania pracy. W efekcie żaden z wątków nie może ruszyć dalej, co prowadzi do zablokowania całego systemu.
- **Wygłodzenie** - jeden z wątków stale nie może uzyskać dostępu do potrzebnych zasobów, ponieważ inne wątki ciągle je zajmują. Wątek może przez to nigdy nie otrzymać szansy na wykonanie swojego zadania.

Dekompozycja Danych

Dekompozycja polega na podziale zadania na mniejsze, niezależne fragmenty, które mogą być wykonywane w osobnych wątkach. Celem tego podejścia jest zwiększenie wydajności oraz przyspieszenie realizacji zadania poprzez wykorzystanie wielu jednostek obliczeniowych jednocześnie.

Rozróżniamy dwa główne sposoby dekompozycji danych:

- Dekompozycja blokowa – dzieląca większe, ciągłe bloki, z których każdy jest przypisany do jednego wątku. Każdy wątek przetwarza swoją część danych niezależnie. Na przykład, jeśli mamy tablicę 1000 elementów i 4 wątki, możemy podzielić ją na 4 bloki po 250 elementów, gdzie każdy wątek zajmuje się jednym blokiem.
- Dekompozycja cykliczna - dane są przydzielane w sposób "przeplatany" - każdy wątek dostaje kolejno co n-tą porcję danych. Na przykład, jeśli mamy tablicę 1000 elementów i 4 wątki, wątek 0 przetwarza elementy 0, 4, 8, itd., wątek 1 elementy 1, 5, 9, itd.

Poniżej znajdują się przykłady dokonania takich partycji na tablicy jednowymiarowej dla następującej, zdefiniowanej struktury `thread_args`:

```
typedef struct
{
  int start, end, stride, id;
  common_args* c_args;
} thread_args;
```

Dekompozycja blokowa:

```
thread_args block_partitioning(int i, int N, int thread_num)
{
  int per_thread = (N + thread_num - 1) / thread_num;

  thread_args arg;
  arg.start = i * per_thread;
  arg.end = (i == thread_num - 1) ? N : (i + 1) * per_thread;
  arg.stride = 1;

  return arg;
}
```

Dekompozycja cykliczna:

```
thread_args cyclic_partitioning(int i, int N, int thread_num)
{
  thread_args arg;
  arg.start = i;
  arg.end = N;
  arg.stride = thread_num;

  return arg;
}
```

Przykład wykorzystania funkcji:

```
for (int i = 0; i < thread_num; i++)
{
     // Cyclic partitioning
     // args[i] = cyclic_partitioning(i, N, thread_num);

     // Block partitioning
     args[i] = block_partitioning(i, N, thread_num);

     // Independent of partitioning model
     args[i].c_args = &c_args;
     args[i].id = i;

     // Make a thread
     int rc = pthread_create(&threads[i], NULL, thread_func, &args[i]);
     if (rc != 0)
          printf("Thread creation failed!\n");
}
```

Przykład aplikacji uzyskanego przedziału w funkcji wątku:

```
void* thread_func(void* arg)
{
    // Cast address to argument type
    thread_args input = *(thread_args*)arg;

    // Allocate memory for the integral
    double* integral = malloc(sizeof(double));
    *integral = 0;

    // Work on the integral
    for(int i = input.start; i < input.end; i += input.stride)
    {
        double x1 = input.c_args->a + i * input.c_args->dx;
        *integral += 0.5 * input.c_args->dx *
          (funkcja(x1) + funkcja(x1 + input.c_args->dx));
    }

    printf("Thread %d has finished it's work\n", input.id);
    pthread_exit((void*)integral);
}
```

Przeprowadzony został test w którym zostały utworzone tablice o wielkościach 1 000 oraz 1 000 000 elementów. Tablice zostały zsumowane na trzy sposoby:

- Sekwencyjnie – wątek główny samodzielnie zsumował wszystkie elementy
- Z użyciem muteksu – tablica została rozdzielona blokowo pomiędzy wątki potomne. Wątki zsumowały jej elementy i używając muteksa dodawały wyniki do wspólnej zmiennej globalnej.
- Z użyciem tablicy wyników – tablica została rozdzielona blokowo pomiędzy wątki potomne. Wątki zsumowały jej elementy i zapisywały wyniki w wyznaczonych dla siebie indeksach w tablicy globalnej, unikając używania muteksu. Wątek nadrzędny następnie zsumował wyniki tej tablicy globalnej.

Testy zostały przeprowadzone dla sytuacji z 2 i 4 wątkami oraz z ustawieniami optymalizacji GCC -03 oraz -g.

W poniższych tabelkach znajdują się wyniki:

Pomiary	Sekwencyjnie [s]	2 wątki mutex[s]	2 wątki tab[s]	4 wątki mutex[s]	4 wątki tab[s]
1	0.000003	0.000344	0.000113	0.000443	0.000174
2	0.000002	0.000233	0.000054	0.000425	0.00085
3	0.000003	0.000369	0.000091	0.000267	0.000057
Średnia	2.66667E-06	0.000315333	0.000086	0.000378333	0.000360333

Tabela 1: 1 000 elementów -g

Pomiary	Sekwencyjnie [s]	2 wątki mutex[s]	2 wątki tab[s]	4 wątki mutex[s]	4 wątki tab[s]
1	0.000001	0.000268	0.000128	0.00135	0.000109
2	0.000001	0.000269	0.0001	0.00166	0.000096
3	0.000001	0.000274	0.00147	0.000373	0.000092
Średnia	0.000001	0.000270333	0.000566	0.001127667	0.000099

Tabela 2: 1 000 elementów -O3

Pomiary	Sekwencyjnie [s]	2 wątki mutex[s]	2 wątki tab[s]	4 wątki mutex[s]	4 wątki tab[s]
1	0.002425	0.001503	0.001221	0.002028	0.001011
2	0.002466	0.001721	0.001287	0.001218	0.000818
3	0.002352	0.001727	0.001431	0.001153	0.000858
Średnia	0.002414333	0.001650333	0.001313	0.001466333	0.000895667

Tabela 3: 1 000 000 elementów -g

Pomiary	Sekwencyjnie [s]	2 wątki mutex[s]	2 wątki tab[s]	4 wątki mutex[s]	4 wątki tab[s]
1	0.001145	0.00128	0.000855	0.001149	0.000703
2	0.001356	0.001549	0.000962	0.001417	0.000978
3	0.001164	0.001333	0.000801	0.001331	0.000943
Średnia	0.001221667	0.001387333	0.000872667	0.001299	0.000874667

Tabela 3: 1 000 000 elementów -O3

Analiza wyników

Dla tablicy o rozmiarze 1 000 elementów wersja sekwencyjna osiągnęła najlepsze wyniki, co jest zgodne z oczekiwaniami – przy tak małej ilości danych narzut związany z synchronizacją w wersji wielowątkowej jest większy niż zysk z równoległego przetwarzania. Wersja z użyciem tablicy wyników była szybsza od wersji z muteksem, ponieważ unikała częstego blokowania i odblokowywania muteksu, co znacząco skróciło czas pracy.

Dla tablicy o rozmiarze 1 000 000 elementów sytuacja wyglądała nieco inaczej. Tutaj sumowanie równoległe przyniosło większe korzyści, ponieważ praca wątków mogła być lepiej zrównoważona i czas wykonywania rozkładał się bardziej równomiernie. Wersja z tablicą wyników nadal przewyższała wersję z muteksem – wyeliminowanie potrzeby synchronizacji wątków pozwoliło uzyskać lepszą wydajność przy jednoczesnym zachowaniu poprawności obliczeń.

Optymalizacja kompilatora (-O3) wpłynęła korzystnie na czas wykonania, co można zaobserwować w wynikach dla obu rozmiarów tablic. Wersja zoptymalizowana była szybsza, ponieważ kompilator mógł zastosować bardziej efektywne instrukcje maszynowe i zminimalizować narzut związany z operacjami wejścia-wyjścia oraz dostępem do pamięci.

Podsumowując, wyniki pokazują, że dla mniejszych tablic podejście sekwencyjne jest efektywniejsze ze względu na minimalny narzut wielowątkowości, natomiast dla dużych tablic lepiej sprawdzają się rozwiązania równoległe, zwłaszcza z użyciem tablicy wyników, która unika kosztownych operacji synchronizacji.

Wnioski

Przeprowadzone laboratorium pozwoliło na zdobycie praktycznych umiejętności w zakresie pracy z wątkami POSIX w języku C, w tym ich tworzenia, synchronizacji, przekazywania danych oraz dekompozycji zadań. Kluczowym elementem było zrozumienie, jak właściwie zarządzać współbieżnością, aby zminimalizować ryzyko błędów oraz osiągnąć lepszą efektywność obliczeń.

Tworzenie i synchronizacja wątków: W trakcie ćwiczeń zastosowano funkcje **pthread_create** oraz **pthread_join**, co pozwoliło na jednoczesne przetwarzanie różnych fragmentów danych. Dzięki **pthread_join** wątki nadrzędne mogły czekać na zakończenie pracy wątków potomnych, co zapewniało spójność wyników.

Rzutowanie i przekazywanie argumentów: Przekazywanie danych do wątków przy pomocy wskaźników pozwala na elastyczną pracę z danymi różnego typu. Użycie struktur, które przechowywały złożone zestawy danych, umożliwiło łatwe przesyłanie parametrów bez konieczności stosowania globalnych zmiennych, co poprawia modularność i bezpieczeństwo kodu.

Muteksy jako narzędzie synchronizacji: Synchronizacja przy pomocy muteksów, jest bardzo ważna w przypadku współdzielenia zmiennych globalnych między wątkami. Mechanizmy te skutecznie zabezpieczyły przed sytuacjami wyścigu (race condition), w których brak odpowiedniego zarządzania dostępem do zasobów prowadziłby do nieprzewidywalnych wyników. Jednak, jak pokazały testy wydajnościowe, narzut czasowy związany z blokowaniem i odblokowywaniem muteksów negatywnie wpływa na wydajność, co jest szczególnie zauważalne przy większej liczbie wątków.

Dekompozycja danych: Wprowadzone metody dekompozycji – blokowa i cykliczna – umożliwiły podział zadania na mniejsze fragmenty przypisane do różnych wątków, co zwiększyło efektywność przetwarzania równoległego. Testy wydajnościowe wykazały, że

przy małych zbiorach danych lepsze rezultaty daje metoda sekwencyjna, natomiast dla dużych zbiorów dekompozycja blokowa lub cykliczna znacząco skraca czas wykonania. Metoda blokowa jest bardziej odpowiednia dla zadań o dużej lokalności pamięci, natomiast metoda cykliczna sprawdza się, gdy dane są bardziej zróżnicowane lub wymagają równomiernego rozkładu obciążenia.

Podsumowanie: Laboratorium pokazało zalety i ograniczenia pracy z wątkami w C oraz wpływu zastosowanych metod synchronizacji i dekompozycji na wydajność programu. W małych zadaniach sekwencyjne przetwarzanie pozostaje optymalne, jednak wraz ze wzrostem liczby danych przetwarzanie wielowątkowe – szczególnie z optymalizacją dekompozycji i unikania kosztownej synchronizacji – daje znaczące korzyści. Poznane narzędzia i techniki są podstawą w tworzenia wydajnych aplikacji równoległych i będą wymagane w przyszłych projektach wymagających efektywnego zarządzania współbieżnością.