Programowanie Równoległe	Kierunek: Informatyka Techniczna	Grupa: 9
Imię i nazwisko: Jakub Świerczyński	Temat: Lab 4, lab 5	Termin oddania: 13.11.2024

Laboratorium 4

Wykonanie

• Przygotowanie środowiska

Utworzono katalog roboczy lab_4 i skopiowano do niego plik pub_sym_1.c, który stanowił punkt wyjścia dla dalszych modyfikacji.

• Analiza początkowego kodu (pub_sym_1.c)

Początkowy kod reprezentował prostą symulację pubu, w której klienci wchodzą do pubu, wybierają kufel, nalewają piwo, piją je, a następnie oddają kufel. Kod nie zawierał mechanizmów synchronizacji, co mogło prowadzić do wyścigów danych, szczególnie podczas pobierania i oddawania kufli.

• Potencjalne błędy:

- Zmiana całkowitej liczby kufli:

Na skutek rywalizacji o kufle między wątkami, całkowita liczba kufli w pubie może ulec zmianie. Może to nastąpić, gdy kilku klientów jednocześnie pobiera lub odkłada kufle bez odpowiedniej synchronizacji.

- Pobranie kufla mimo braku wolnych:

Klient może pobrać kufel, mimo że nie ma już wolnych kufli w pubie. Dzieje się tak, gdy brak jest sprawdzenia dostępności kufli przed ich pobraniem lub gdy to sprawdzenie nie jest odpowiednio zsynchronizowane.

```
Klient 44 oddaje kufel. Kufli dostępnych: 45
Klient 44 wychodzi z pubu
Klient 46 oddaje kufel. Kufli dostępnych: 46
Klient 46 wychodzi z pubu
Klient 45 oddaje kufel. Kufli dostępnych: 47
Klient 45 wychodzi z pubu
Klient 47 oddaje kufel. Kufli dostępnych: 48
Klient 47 wychodzi z pubu
Klient 48 oddaje kufel. Kufli dostępnych: 49
Klient 48 wychodzi z pubu
Klient 49 oddaje kufel. Kufli dostępnych: 50
Klient 49 wychodzi z pubu
Zamykamy pub! Dostępne kufle na koniec: 50
```

```
Liczba klientow: 50
Liczba kufli: 3
```

Przykład błędu w którym występuje rywalizacja o kufle

```
Klient 40 pobiera kufel. Kufli pozostało: 9
Klient 40 nalewa piwo
Klient 41 pobiera kufel. Kufli pozostało: 8
Klient 41 nalewa piwo
Klient 42 pobiera kufel. Kufli pozostało: 7
Klient 42 nalewa piwo
Klient 43 pobiera kufel. Kufli pozostało: 6
Klient 43 nalewa piwo
Klient 44 pobiera kufel. Kufli pozostało: 5
Klient 44 nalewa piwo
Klient 45 pobiera kufel. Kufli pozostało: 4
Klient 45 nalewa piwo
```

• Najprostszą reprezentacją pozwalającą na rozwiązanie problemu bezpiecznego korzystania z kufli jest wprowadzenie globalnej zmiennej liczba_wolnych_kufli, która będzie przechowywać aktualną liczbę dostępnych kufli w pubie. Dzięki synchronizacji dostępu do tej zmiennej za pomocą mutexu, możemy zapewnić, że tylko jeden wątek na raz modyfikuje jej wartość, co eliminuje wyścigi danych.

```
int liczba_wolnych_kufli;
```

• Dodanie sprawdzeń potencjalnych błędów

Przed rozpoczęciem pracy pubu zapamiętujemy początkową liczbę kufli. Po zakończeniu pracy wszystkich wątków porównujemy tę wartość z aktualną liczbą kufli.

```
int poczatkowa_liczba_kufli;
```

```
liczba_wolnych_kufli = l_kf;
```

```
if (liczba_wolnych_kufli == poczatkowa_liczba_kufli) {
        printf("\nLiczba kufli na końcu jest zgodna z początkową: %d\n",
liczba_wolnych_kufli);
    } else {
        printf("\nBłąd: liczba kufli na końcu (%d) różni się od początkowej (%d)\n",
liczba_wolnych_kufli, poczatkowa_liczba_kufli);
    }
```

• Sprawdzenie pobrania kufla mimo braku wolnych (w funkcji watek_klient)

Zaraz po zmniejszeniu liczby wolnych kufli sprawdzamy, czy wartość tej zmiennej nie jest ujemna. Jeśli jest, oznacza to, że klient pobrał kufel mimo braku dostępnych.

```
if (liczba_wolnych_kufli < 0) {
            printf("\nBłąd: Klient %d pobrał kufel mimo braku wolnych kufli!\n", moj_id);
            liczba_wolnych_kufli++;
```

• Wprowadzenie mechanizmów synchronizacji za pomocą mutexów

Aby zapewnić bezpieczny dostęp do wspólnej zmiennej liczba_wolnych_kufli, zastosowaliśmy mutex mutex_kufle.

```
pthread_mutex_t mutex_kufle = PTHREAD_MUTEX_INITIALIZER;
```

```
for (i = 0; i < ILE_MUSZE_WYPIC; i++) {
        pthread_mutex_lock(&mutex_kufle); // Zablokuj dostęp do kufli
        if (liczba_wolnych_kufli > 0) {
            liczba_wolnych_kufli--;
            printf("\nKlient %d, pobieram kufel (pozostało %d wolnych kufli)\n", moj_id, liczba_wolnych_kufli);
            pthread_mutex_unlock(&mutex_kufle); // Zwolnij dostęp po pobraniu

            // Nalewanie i picie
            printf("\nKlient %d, nalewam z kranu\n", moj_id);
            usleep(30000); // Nalewanie piwa
            printf("\nKlient %d, pije\n", moj_id);
            nanosleep((struct timespec[]){{0, 50000000L}}, NULL); // Picie piwa

            // Odkładanie kufla
            pthread_mutex_lock(&mutex_kufle); // Zablokuj dostęp do kufli
            liczba_wolnych_kufli++;
            printf("\nKlient %d, odkladam kufel (wolnych kufli %d)\n", moj_id, liczba_wolnych_kufli);
            pthread_mutex_unlock(&mutex_kufle); // Zwolnij dostęp po odłożeniu
        } else {
            printf("\nKlient %d, nie ma wolnych kufli, czekam\n", moj_id);
            pthread_mutex_unlock(&mutex_kufle); // Zwolnij dostęp, jeśli brak kufli
            usleep(10000); // Krótkie czekanie, jeśli brak kufli
        }
    }
```

• Implementacja aktywnego czekania (busy waiting)

Aby uniknąć sytuacji, w której klient pobiera kufel mimo braku dostępnych, wprowadziliśmy mechanizm aktywnego czekania.

```
void *watek_klient(void *arg_wsk) {
    int moj_id = *((int *)arg_wsk);
    int i;
    printf("\nKlient %d, wchodzę do pubu\n", moj_id);
    for (i = 0; i < ILE_MUSZE_WYPIC; i++) {
        int success = 0;
        do {
            pthread_mutex_lock(&mutex_kufle);
            if (liczba_wolnych_kufli > 0) {
                liczba_wolnych_kufli--;
                printf("\nKlient %d, pobieram kufel (pozostało %d wolnych kufli)\n", moj_id, liczba_wolnych_kufli);
                success = 1;
            }
            pthread_mutex_unlock(&mutex_kufle);

            if (!success) {
                printf("\nKlient %d, czekam na wolny kufel\n", moj_id);
                usleep(10000);
            }
        } while (!success);
        printf("\nKlient %d, nalewam z kranu\n", moj_id);
        usleep(30000);
```

```
        printf("\nKlient %d, pije\n", moj_id);
        nanosleep((struct timespec[]){{0, 50000000L}}, NULL);
        pthread_mutex_lock(&mutex_kufle);
        liczba_wolnych_kufli++;
        printf("\nKlient %d, odkladam kufel (wolnych kufli %d)\n", moj_id, liczba_wolnych_kufli);
        pthread_mutex_unlock(&mutex_kufle);
    }
    printf("\nKlient %d, wychodzę z pubu\n", moj_id);
    return NULL;
}
```

Wnioski

• Przy uruchomieniu programu z liczbą kufli większą od liczby klientów i zastosowaniu mutexów, program działał poprawnie. Liczba kufli na końcu była zgodna z początkową, a klienci nie pobierali kufli przy braku dostępnych.

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab4# ./mutexy_pub

Liczba klientow: 50

Liczba kufli: 5

Otwieramy pub (simple)!

Liczba wolnych kufli 5
```

```
Klient 3, wychodzę z pubu

Klient 0, odkladam kufel (wolnych kufli 3)

Klient 0, wychodzę z pubu

Klient 4, odkladam kufel (wolnych kufli 4)

Klient 4, wychodzę z pubu

Klient 2, odkladam kufel (wolnych kufli 5)

Klient 2, wychodzę z pubu

Zamykamy pub!

Liczba kufli na końcu jest zgodna z początkową: 5
```

• Jak wygląda rozwiązanie problemu bezpiecznego korzystania z kufli? Jak połączyć je ze sprawdzeniem dostępności kufli?

- Rozwiązanie polega na zastosowaniu mutexu do ochrony sekcji krytycznej, w której wątki sprawdzają dostępność kufli i modyfikują ich liczbę. Wątek blokuje mutex, sprawdza, czy liczba_wolnych_kufli jest większa od zera, a następnie ją dekrementuje. Jeśli kufli brak, wątek zwalnia mutex i aktywnie czeka, aż kufel się zwolni.

• Jaka jest wada rozwiązania z wykorzystaniem tylko mutexów?

- Wadą jest zastosowanie aktywnego czekania (busy waiting), które polega na ciągłym sprawdzaniu warunku w pętli. Powoduje to marnowanie zasobów procesora, ponieważ wątki zużywają czas CPU na nieustanne sprawdzanie dostępności kufli, zamiast efektywnie oczekiwać na ich zwolnienie. Jeśli w miejscu do_something_else_or_nothing(); nie ma sensownych operacji do wykonania, zasoby sprzętowe są marnowane.

Laboratorium 5

Wykonanie

• Wzorzec zrównoleglenia pętli

Fragment kodu funkcji wątku z dekompozycją blokową:

```
void *suma_w( void *arg_wsk){

  int i, j, moj_id;

  double moja_suma=0;

  moj_id = *( (int *) arg_wsk );

  j=ceil( (float)ROZMIAR/LICZBA_W );
  if(j*LICZBA_W!=ROZMIAR) { printf("Error! Exiting.\n"); exit(0);}

  for( i = j*moj_id; i < j*(moj_id+1); i++){
    moja_suma += tab[i];
  }

  pthread_mutex_lock( &muteks );
  suma += moja_suma;
  pthread_mutex_unlock( &muteks );

  pthread_exit( NULL );

}
```

Omówienie działania wątku:

- Dekompozycja blokowa: Zakładamy, że tablica jest podzielona na równe bloki, a każdy wątek przetwarza jeden blok.

- Obliczanie zakresu indeksów.

- Sumowanie: Wątek sumuje elementy tablicy.

- Sekcja krytyczna: Dodawanie lokalnej sumy moja_suma do globalnej zmiennej suma odbywa się w sekcji krytycznej zabezpieczonej mutexem.

•Jak wygląda prosty wzorzec zrównoleglenia pętli?

Prosty wzorzec zrównoleglenia pętli polega na podziale zakresu iteracji pętli między wątki. Każdy wątek wykonuje swoją część pętli niezależnie, a następnie wyniki są łączone.

•Rodzaj dekompozycji pętli:

Dekompozycja blokowa: Zakres iteracji jest dzielony na bloki o zbliżonym rozmiarze. Każdy wątek przetwarza jeden blok.

Testy wydajności

Wyniki dla ROZMIAR = 1000 i LICZBA_W = 2:

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab5/zad1/pthreads_suma# ./pthreads_suma
Obliczenia sekwencyjne
suma = 500.500000
Czas obliczen sekwencyjnych = 0.000001
Poczatek tworzenia watkow
suma = 500.500000
Czas obliczen 2 wątków = 0.000213
Poczatek tworzenia watkow
suma = 500.500000
Czas obliczen 2 wątków (globalna tablica zamiast mutex'a) = 0.000117
```

Wyniki dla ROZMIAR = 100000000 i LICZBA_W = 2:

```
root@DESKTOP-MEAKJDN:/mnt/d/agh/Rownolegle/lab5/zad1/pthreads_suma# ./pthreads_suma
Obliczenia sekwencyjne
suma = 50000000.500000
Czas obliczen sekwencyjnych = 0.099436
Poczatek tworzenia watkow
suma = 50000000.500000
Czas obliczen 2 wątków = 0.050106
Poczatek tworzenia watkow
suma = 50000000.500000
Czas obliczen 2 wątków (globalna tablica zamiast mutex'a) = 0.050082
```

Wnioski

• Przy małym rozmiarze tablicy narzut związany z tworzeniem wątków i synchronizacją przewyższa korzyści z równoległości.

• Przy dużym rozmiarze tablicy wersja równoległa znacząco skraca czas obliczeń.

• Wersja bez mutexa (z tablicą wyników lokalnych) jest nieco szybsza niż z mutexem.

• Uruchomienie obliczanie_calki.c

Wysokość trapezu = 0.02, ilość wątków = 2

```
Podaj wysokość pojedynczego trapezu:  0.02

Podaj liczbę wątków:  2

Poczatek obliczeń sekwencyjnych
Obliczona liczba trapezów: N = 158, dx_adjust = 0.019883

Koniec obliczen sekwencyjnych
        Czas wykonania 0.000034.        Obliczona całka = 1.999934107318307

Poczatek obliczeń równoległych (zrównoleglenie pętli)
Obliczona liczba trapezów: N = 158, dx_adjust = 0.019883

Koniec obliczen  równoległych (zrównoleglenie pętli)
        Czas wykonania 0.000601.        Obliczona całka = 1.999934107318307
```

Wysokość trapezu = 0.002, ilość wątków = 2

```
Podaj wysokość pojedynczego trapezu:  0.002

Podaj liczbę wątków:  2

Poczatek obliczeń sekwencyjnych
Obliczona liczba trapezów: N = 1571, dx_adjust = 0.002000

Koniec obliczen sekwencyjnych
        Czas wykonania 0.000057.        Obliczona całka = 1.999999333506139

Poczatek obliczeń równoległych (zrównoleglenie pętli)
Obliczona liczba trapezów: N = 1571, dx_adjust = 0.002000

Koniec obliczen  równoległych (zrównoleglenie pętli)
        Czas wykonania 0.000292.        Obliczona całka = 1.999999333506137
```

Wysokość trapezu = 0.0002, ilość wątków = 2

```
Podaj wysokość pojedynczego trapezu:  0.0002

Podaj liczbę wątków:  2

Poczatek obliczeń sekwencyjnych
Obliczona liczba trapezów: N = 15708, dx_adjust = 0.000200

Koniec obliczen sekwencyjnych
        Czas wykonania 0.000276.        Obliczona całka = 1.999999993333371

Poczatek obliczeń równoległych (zrównoleglenie pętli)
Obliczona liczba trapezów: N = 15708, dx_adjust = 0.000200

Koniec obliczen  równoległych (zrównoleglenie pętli)
        Czas wykonania 0.000406.        Obliczona całka = 1.999999993333362
```

Przy malejącej wartości dx i rosnącej liczbie trapezów N, wynik całkowania zbiega do wartości 2.0, co jest zgodne z analityczną wartością całki z funkcji sin(x) w przedziale [0, π].

• Kod funkcji calka_zrownoleglenie_petli

```
double calka_zrownoleglenie_petli(double a, double b, double dx, int l_w){

  int N = ceil((b-a)/dx);
  double dx_adjust = (b-a)/N;

  printf("Obliczona liczba trapezów: N = %d, dx_adjust = %lf\n", N, dx_adjust);

  // Inicjalizacja mutexu
  pthread_mutex_init(&mutex_calka, NULL);

  // Przekazanie wartości do zmiennych globalnych
    a_global = a;
    b_global = b;
    dx_global = dx_adjust;
    N_global = N;
    l_w_global = l_w;

  // Alokacja pamięci
    pthread_t *threads = malloc(l_w * sizeof(pthread_t));
    int *thread_ids = malloc(l_w * sizeof(int));

  // Tworzenie wątków
    for (int i = 0; i < l_w; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, calka_fragment_petli_w, (void *)&thread_ids[i]);
    }

  // Oczekiwanie na zakończenie pracy wątków i zebranie wyników
    for (int i = 0; i < l_w; i++) {
        pthread_join(threads[i], NULL);
    }

    // Zwolnienie pamięci
    pthread_mutex_destroy(&mutex_calka);
    free(threads);
    free(thread_ids);


  return(calka_global);
}
```

• Dekompozycja blokowa

```
  // dekompozycja blokowa
  int my_start = my_id * (N_global / l_w_global);
  int my_end = (my_id + 1) * (N_global / l_w_global);
  int my_stride = 1;
```

• Wynik programu dla dekompozycji blokowej.

```
Podaj wysokość pojedynczego trapezu:  0.002

Podaj liczbę wątków:  2

Poczatek obliczeń sekwencyjnych
Obliczona liczba trapezów: N = 1571, dx_adjust = 0.002000

Koniec obliczen sekwencyjnych
        Czas wykonania 0.000056.        Obliczona całka = 1.999999333506139

Poczatek obliczeń równoległych (zrównoleglenie pętli)
Obliczona liczba trapezów: N = 1571, dx_adjust = 0.002000

Koniec obliczen  równoległych (zrównoleglenie pętli)
        Czas wykonania 0.000280.        Obliczona całka = 1.999999333506137
```

W obu wersjach (sekwencyjnej i równoległej) liczba trapezów N oraz wysokość dx_adjust są identyczne. Dzięki temu wyniki obu wersji powinny być identyczne (z dokładnością do błędów numerycznych).

Wnioski

• W obu wersjach wykonywane są te same obliczenia na tych samych danych wejściowych. Dekompozycja pętli nie wpływa na logikę obliczeń, a jedynie rozdziela pracę między wątki. Ponieważ suma wszystkich wyników cząstkowych jest taka sama jak w wersji sekwencyjnej, ostateczny wynik jest identyczny. Niewielkie różnice mogą wynikać z kolejności sumowania liczb zmiennoprzecinkowych, co wpływa na dokładność numeryczną.

• Wady wersji ze zrównolegleniem pętli:
- Złożoność implementacji: Konieczność zarządzania wątkami, mutexami i zmiennymi globalnymi zwiększa złożoność kodu.

- Wykorzystanie zmiennych globalnych: Przekazywanie danych poprzez zmienne globalne jest niekorzystne z punktu widzenia inżynierii oprogramowania, utrudnia debugowanie i konserwację kodu.

- Narzut synchronizacji: Użycie mutexu do ochrony sekcji krytycznej może wpływać na wydajność, zwłaszcza przy dużej liczbie wątków.

- Nierównomierne obciążenie wątków: W zależności od dekompozycji (cykliczna vs blokowa) wątki mogą być nierównomiernie obciążone, co wpływa na efektywność równoległości.

- Skalowalność: Przy bardzo dużej liczbie wątków narzut związany z zarządzaniem wątkami i synchronizacją może przewyższyć korzyści z równoległego przetwarzania.