Laboratorium nr. 4, laboratorium nr. 5	
Imię i nazwisko: Miłosz Dębowski	Kierunek: Informatyka Techniczna
Numer albumu: 415045	Gr. Lab.: 8

Laboratorium nr. 4

1. Cel: opanowanie umiejętności synchronizacji wątków oraz bezpiecznego dostępu do współdzielonych zasobów przy wykorzystaniu muteksów. Celem było stworzenie programu symulującego pub, w którym klienci (jako wątki) konkurują o kufle do picia piwa.
2. Opis symulacji
    a. Reprezentacja pubu: określenie, że pub zawiera ograniczoną liczbę kufli (l_kf) i klientów (l_kl), którzy chcą wypić określoną liczbę litrów piwa (ile_musze_wypic).
    b. Klienci jako wątki: wyjaśnienie, że każdy klient jest reprezentowany przez oddzielny wątek i symulacja przebiega w sposób równoległy, aby zasymulować naturalną rywalizację klientów o kufle.
    c. Scenariusz działania: opis działania, w którym klient pobiera kufel, napełnia go, pije piwo, a następnie oddaje kufel. Proces jest powtarzany do momentu, aż klient wypije wymaganą ilość piwa.
3. Problemy do rozwiązania
    - Wyścigi przy dostępie do kufli: sytuacja, w której dwóch klientów jednocześnie próbuje pobrać kufel, co skutkuje błędami. Omówienie problemów wynikających z braku mechanizmu wzajemnego wykluczania przy pobieraniu kufla.
    - Przekroczenie limitu kufli: opis błędu, kiedy w pubie może się pojawić więcej kufli niż na początku (błąd wynikający z braku synchronizacji).
    - Mechanizm kontrolny: wskazanie potrzeby weryfikacji początkowej i końcowej liczby kufli, aby upewnić się, że nie wystąpiły błędy związane z równoczesnym dostępem.
4. Pierwsza wersja programu (brak zabezpieczeń/brak synchronizacji)

- Kod fragmentów pobierania, oddawania kufla i sprawdzenie ilości kufla: przedstawienie pierwszej wersji kodu, gdzie pobranie i oddanie kufla nie są zabezpieczone.

```
void * watek_klient (void * arg);
int liczba_kufli=0;
```

```
for(i=0; i<ile_musze_wypic; i++){
    liczba_kufli--;
    printf("\nKlient %d, wybieram kufel\n", moj_id);

    j=0;
    printf("\nKlient %d, nalewam z kranu %d\n", moj_id, j);
    usleep(30);

    printf("\nKlient %d, pije\n", moj_id);
    nanosleep((struct timespec[]){{0, 50000000L}}, NULL);
    liczba_kufli++;;
    printf("\nKlient %d, odkladam kufel\n", moj_id);

  }
```

- Wyniki uruchomienia:

```
Klient 1, wychodzÄ z pubu; wykonana praca 0

Klient 2, odkladam kufel

Klient 2, wychodzÄ z pubu; wykonana praca 0

Zamykamy pub!
Liczba kufli się zgadza
Liczba kufli: 2
neox@DESKTOP-CVRD880:/mnt/d/AGH/Programowanie równoległe/lab_4$
```

- Analiza błędów:

  W przypadku kiedy jest 3 klientów a 2 kufle liczba kufli na koniec działania programu się nie zgadza. Wynika to z braku zabezpieczenia pobierania i oddawania kufla, przez co wątki pobierają ten sam kufel.

5. Druga wersja programu (zabezpieczenie pobierania i oddawania kufli/ synchonizacja za pomocą mutexa)
    - Kod fragmentów pobierania i oddawania kufla: przedstawienie wersji kodu, gdzie pobranie i oddanie kufla są zabezpieczone.

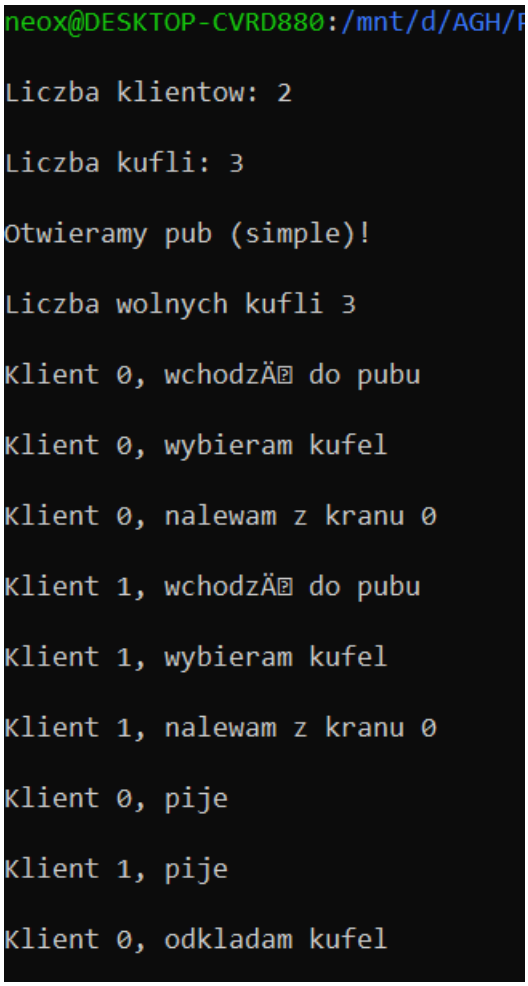
```
void * watek_klient (void * arg);
pthread_mutex_t mutex;
int liczba_kufli=0;
```

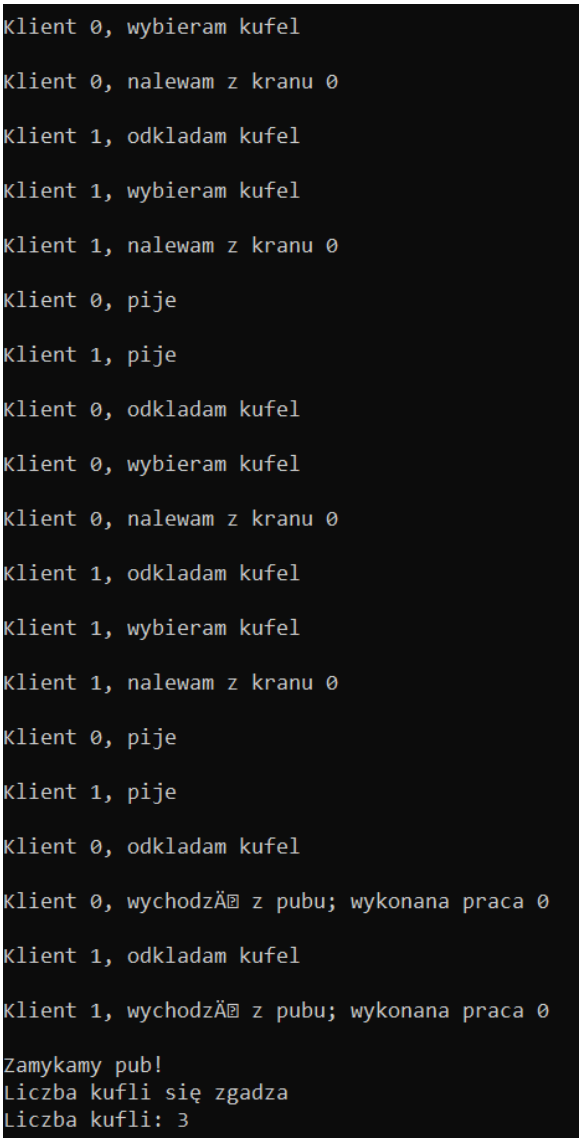
```
pthread_mutex_init(&mutex, NULL);
```

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

```
for(i=0; i<ile_musze_wypic; i++){
    pthread_mutex_lock(&mutex);
    liczba_kufli--;
    printf("\nKlient %d, wybieram kufel\n", moj_id);
    pthread_mutex_unlock(&mutex);
    j=0;
    printf("\nKlient %d, nalewam z kranu %d\n", moj_id, j);
    usleep(30);

    printf("\nKlient %d, pije\n", moj_id);
    nanosleep((struct timespec[]){{0, 50000000L}}, NULL);


    pthread_mutex_lock(&mutex);
    liczba_kufli++;;
    printf("\nKlient %d, odkladam kufel\n", moj_id);

  }
```

- Wyniki uruchomienia
    a) Liczba kufli większa niż liczba klientów





b) Liczba kufli mniejsza niż liczba klientów

```
Liczba klientow: 5
Liczba kufli: 2
Otwieramy pub (simple)!
Liczba wolnych kufli 2
Klient 0, wchodzÄ pub
Klient 0, wybieram kufel
Klient 0, nalewam z kranu 0
Klient 1, wchodzÄ pub
Klient 1, wybieram kufel
Klient 1, nalewam z kranu 0
Klient 2, wchodzÄ pub
Klient 2, wybieram kufel
Klient 2, nalewam z kranu 0
Klient 0, pije
Klient 1, pije
Klient 2, pije
Klient 3, wchodzÄ pub
Klient 3, wybieram kufel
Klient 3, nalewam z kranu 0
Klient 4, wchodzÄ pub
Klient 4, wybieram kufel
Klient 4, nalewam z kranu 0
```

```
Klient 3, pije
Klient 4, pije
Klient 0, odkladam kufel
Klient 0, wybieram kufel
Klient 0, nalewam z kranu 0
Klient 2, odkladam kufel
Klient 2, wybieram kufel
Klient 2, nalewam z kranu 0
Klient 1, odkladam kufel
Klient 1, wybieram kufel
Klient 1, nalewam z kranu 0
Klient 2, pije
Klient 3, odkladam kufel
Klient 1, pije
Klient 0, pije
Klient 3, wybieram kufel
Klient 3, nalewam z kranu 0
Klient 4, odkladam kufel
Klient 4, wybieram kufel
Klient 4, nalewam z kranu 0
Klient 3, pije
Klient 4, pije
```

```
Klient 2, odkladam kufel
Klient 0, odkladam kufel
Klient 0, wybieram kufel
Klient 0, nalewam z kranu 0
Klient 2, wybieram kufel
Klient 2, nalewam z kranu 0
Klient 1, odkladam kufel
Klient 1, wybieram kufel
Klient 1, nalewam z kranu 0
Klient 0, pije
Klient 2, pije
Klient 1, pije
Klient 4, odkladam kufel
Klient 4, wybieram kufel
Klient 4, nalewam z kranu 0
Klient 3, odkladam kufel
Klient 3, wybieram kufel
Klient 3, nalewam z kranu 0
Klient 4, pije
Klient 3, pije
Klient 0, odkladam kufel
Klient 0, wychodzÄ z pubu; wykonana praca 0
```

```
Klient 0, wychodzÄ z pubu; wykonana praca 0
Klient 1, odkladam kufel
Klient 1, wychodzÄ z pubu; wykonana praca 0
Klient 2, odkladam kufel
Klient 2, wychodzÄ z pubu; wykonana praca 0
Klient 4, odkladam kufel
Klient 4, wychodzÄ z pubu; wykonana praca 0
Klient 3, odkladam kufel
Klient 3, wychodzÄ z pubu; wykonana praca 0
Zamykamy pub!
Liczba kufli się zgadza
Liczba kufli: 2
```

- Analiza błędów
  Klieneci pobierają kufel mimo, że ich nie ma. Powodem jest brak sprawdzenia ilości kufli przed pobraniem kufla oraz brak zastosowania metody aktywnego czekania.

6. Implementacja aktywnego czekania (busy waiting)
   - Kod implementacji aktywnego czekania

```
  for(i=0; i<ile_musze_wypic; i++){
    pthread_mutex_lock(&mutex);
    while (liczba_kufli <= 0) {
        printf("Klient %d, Brak wolnych kufli. Czekam\n", moj_id);
        pthread_mutex_unlock(&mutex);
        usleep(100000); //klient czeka przed ponownym sprawdzeniem czy
jest kufel
        //usleep(1);
        pthread_mutex_lock(&mutex);
    }
    liczba_kufli--;//klient bierze kufel
    printf("\nKlient %d, wybieram kufel. Liczba pozostałych kufli:
%d\n", moj_id, liczba_kufli);
    pthread_mutex_unlock(&mutex);
```

   - Wyniki uruchomienia (więcej klientów niż kufli)

```
Liczba klientow: 4

Liczba kufli: 2

Otwieramy pub (simple)!

Liczba wolnych kufli 2

Klient 0, wchodzę do pubu

Klient 0, wybieram kufel. Liczba pozostałych kufli: 1

Klient 1, wchodzę do pubu

Klient 2, wchodzę do pubu

Klient 3, wchodzę do pubu

Klient 1, wybieram kufel. Liczba pozostałych kufli: 0

Klient 1, nalewam z kranu 0

Klient 0, nalewam z kranu 0
Klient 3, Brak wolnych kufli. Czekam
Klient 2, Brak wolnych kufli. Czekam

Klient 1, pije

Klient 0, pije

Klient 1, odkladam kufel. Liczba dostępnych kufli: 1

Klient 1, wybieram kufel. Liczba pozostałych kufli: 0
```

```
Klient 1, nalewam z kranu 0

Klient 0, odkladam kufel. Liczba dostępnych kufli: 1

Klient 0, wybieram kufel. Liczba pozostałych kufli: 0

Klient 0, nalewam z kranu 0

Klient 1, pije

Klient 0, pije
Klient 3, Brak wolnych kufli. Czekam
Klient 2, Brak wolnych kufli. Czekam

Klient 1, odkladam kufel. Liczba dostępnych kufli: 1

Klient 1, wybieram kufel. Liczba pozostałych kufli: 0

Klient 1, nalewam z kranu 0

Klient 0, odkladam kufel. Liczba dostępnych kufli: 1

Klient 0, wybieram kufel. Liczba pozostałych kufli: 0

Klient 0, nalewam z kranu 0

Klient 1, pije

Klient 0, pije

Klient 0, odkladam kufel. Liczba dostępnych kufli: 1

Klient 0, wychodzę z pubu

Klient 1, odkladam kufel. Liczba dostępnych kufli: 2

Klient 1, wychodzę z pubu

Klient 3, wybieram kufel. Liczba pozostałych kufli: 1

Klient 3, nalewam z kranu 0

Klient 2, wybieram kufel. Liczba pozostałych kufli: 0
```

```
Klient 2, nalewam z kranu 0

Klient 2, pije

Klient 3, pije

Klient 3, odkladam kufel. Liczba dostępnych kufli: 1

Klient 3, wybieram kufel. Liczba pozostałych kufli: 0

Klient 3, nalewam z kranu 0

Klient 2, odkladam kufel. Liczba dostępnych kufli: 1

Klient 2, wybieram kufel. Liczba pozostałych kufli: 0

Klient 2, nalewam z kranu 0

Klient 2, pije

Klient 3, pije

Klient 2, odkladam kufel. Liczba dostępnych kufli: 1

Klient 2, wybieram kufel. Liczba pozostałych kufli: 0

Klient 2, nalewam z kranu 0

Klient 3, odkladam kufel. Liczba dostępnych kufli: 1

Klient 3, wybieram kufel. Liczba pozostałych kufli: 0

Klient 3, nalewam z kranu 0

Klient 2, pije

Klient 3, pije

Klient 3, odkladam kufel. Liczba dostępnych kufli: 1

Klient 3, wychodzę z pubu

Klient 2, odkladam kufel. Liczba dostępnych kufli: 2
```

```
Klient 2, odkladam kufel. Liczba dostępnych kufli: 2

Klient 2, wychodzę z pubu

Zamykamy pub!
Liczba kufli się zgadza
Liczba kufli: 2
```

- Analiza kodu
  Dzięki zastosowaniu metody aktywnego czekania (busy waiting) w przypadku braku kufli klient czeka na zwolnienie kufla

7. Implementacja funkcji trylock()
   a) Kod implementacji funkcji trylock()

```
void * watek_klient(void * arg_wsk) {
    int moj_id = *((int *)arg_wsk);
    int i;
    long int wykonana_praca = 0;

    printf("\nKlient %d, wchodzę do pubu\n", moj_id);

    for (i = 0; i < ILE_MUSZE_WYPIC; i++) {
        while (1) {
            if (pthread_mutex_trylock(&mutex_kufli) == 0) {
                if (liczba_kufli > 0) {
                    liczba_kufli--;
                    printf("\nKlient %d, wybieram kufel. Liczba kufli:
%d\n", moj_id, liczba_kufli);
                    pthread_mutex_unlock(&mutex_kufli);
                    break;
                }
                pthread_mutex_unlock(&mutex_kufli);
            }
            wykonana_praca++;
            usleep(100000);
        }
```

   b) Wyniki uruchomienia:

```
Liczba klientow: 3

Liczba kufli: 1

Liczba kranow: 1

Otwieramy pub!

Liczba wolnych kufli: 1
Liczba dostępnych kranów: 1

Klient 0, wchodzę do pubu

Klient 0, wybieram kufel. Liczba kufli: 0

Klient 0, nalewam z kranu. Liczba dostępnych kranów: 0

Klient 1, wchodzę do pubu

Klient 2, wchodzę do pubu

Klient 0, zwalniam kran. Liczba dostępnych kranów: 1

Klient 0, pije

Klient 0, odkladam kufel. Liczba kufli: 1

Klient 0, wybieram kufel. Liczba kufli: 0

Klient 0, nalewam z kranu. Liczba dostępnych kranów: 0

Klient 0, zwalniam kran. Liczba dostępnych kranów: 1

Klient 0, pije

Klient 0, odkladam kufel. Liczba kufli: 1

Klient 0, wybieram kufel. Liczba kufli: 0

Klient 0, nalewam z kranu. Liczba dostępnych kranów: 0
```

```
Klient 0, zwalniam kran. Liczba dostępnych kranów: 1

Klient 0, pije

Klient 0, odkladam kufel. Liczba kufli: 1

Klient 0, wychodzę z pubu, wykonana praca: 0

Klient 2, wybieram kufel. Liczba kufli: 0

Klient 2, nalewam z kranu. Liczba dostępnych kranów: 0

Klient 2, zwalniam kran. Liczba dostępnych kranów: 1

Klient 2, pije

Klient 2, odkladam kufel. Liczba kufli: 1

Klient 2, wybieram kufel. Liczba kufli: 0

Klient 2, nalewam z kranu. Liczba dostępnych kranów: 0

Klient 2, zwalniam kran. Liczba dostępnych kranów: 1

Klient 2, pije

Klient 2, odkladam kufel. Liczba kufli: 1

Klient 2, wybieram kufel. Liczba kufli: 0

Klient 2, nalewam z kranu. Liczba dostępnych kranów: 0

Klient 2, zwalniam kran. Liczba dostępnych kranów: 1

Klient 2, pije

Klient 2, odkladam kufel. Liczba kufli: 1

Klient 2, wychodzę z pubu, wykonana praca: 3

Klient 1, wybieram kufel. Liczba kufli: 0

Klient 1, nalewam z kranu. Liczba dostępnych kranów: 0
```

```
Klient 1, zwalniam kran. Liczba dostępnych kranów: 1

Klient 1, pije

Klient 1, odkladam kufel. Liczba kufli: 1

Klient 1, wybieram kufel. Liczba kufli: 0

Klient 1, nalewam z kranu. Liczba dostępnych kranów: 0

Klient 1, zwalniam kran. Liczba dostępnych kranów: 1

Klient 1, pije

Klient 1, odkladam kufel. Liczba kufli: 1

Klient 1, wychodzę z pubu, wykonana praca: 6

Zamykamy pub!
Liczba kufli się zgadza
Liczba kufli: 1
```

8. Wnioski
   Muteksy pozwalają kontrolować kolejność wykonywania się wątków działających równolegle i niezależnie od siebie oraz umożliwiają synchronizację dostępu do tzw. sekcji krytycznych — fragmentów kodu, których nie mogą wykonywać jednocześnie różne wątki. Każdy muteks musi mieć właściciela, a jego nazwa musi być unikalna. Jeśli jakieś zadanie zablokuje muteks, to tylko ono może go odblokować, co pozwala eliminować problem niespójności danych. Wadą stosowania muteksów jest jednak ograniczenie pełnego wykorzystania mocy obliczeniowej procesora, ponieważ blokują one części kodu, co wpływa na wydajność systemu.

Laboratorium nr. 5

1. Cel ćwiczenia: Nabycie umiejętności tworzenia i implementacji programów równoległych w środowisku Pthreads
2. Wykonanie zadania nr.1

   - Utworzenie katalogu roboczego
   - Rozpakowanie archiwum
   - Analiza kodu
   - Uruchomienie programów
   - Analiza wyników dla ROZMIAR = 1000000

3. Wyniki z zadania nr.1

	Sekwencyjne	2 wątki	2 wątki (no mutex)	4 wątki	4 wątki (no mutex)	8 wątków	8 wątków (no mutex)
Czas obliczeń	0.003130	0.002067	0.001947	0.001626	0.001883	0.001447	0.001416
	0.003554	0.002276	0.001606	0.001463	0.001297	0.002570	0.001293
	0.003066	0.001990	0.001665	0.001494	0.001410	0.001134	0.001552
Średnia:	0.00325	0.002111	0.001739	0.001527	0.001530	0.001717	0.001420

Wyniki wyraźnie wskazują, że wzrost liczby wątków ogólnie skraca czas obliczeń. Przykładowo, średni czas obliczeń dla 2 wątków z mutexem wynosi 0,002111 s, podczas gdy dla 4 wątków spada do 0,001527 s, co pokazuje wyraźną poprawę. Zwiększenie liczby wątków do 8 również skraca czas obliczeń, jednak wzrost wydajności jest mniej wyraźny, co może sugerować, że dalsze zwiększanie liczby wątków ma coraz mniejszy wpływ na skrócenie czasu obliczeń.

Wyniki pokazują istotne różnice między czasami obliczeń dla wątków korzystających z mutexów i tych, które ich nie używają. Dla 2 wątków obliczenia bez mutexa są średnio o około 18% szybsze (średnia 0,001739 s dla wątków bez mutexa wobec 0,002111 s dla wątków z mutexem). Podobny trend występuje przy większej liczbie wątków: dla 4 wątków czas bez mutexa wynosi 0,001530 s, co jest nieco szybsze od 0,001527 s z mutexem. Można zaobserwować, że mutexy wprowadzają dodatkowe narzuty, ale zapewniają stabilność i synchronizację dostępu do zasobów, co może być kluczowe w aplikacjach wymagających wysokiej spójności danych.

Z wyników wynika, że czas obliczeń nie zmniejsza się proporcjonalnie do liczby wątków, co jest szczególnie widoczne przy przejściu z 4 do 8 wątków, gdzie różnica w czasie jest minimalna. Oznacza to, że przy zwiększaniu liczby wątków efektywność może się zmniejszać, prawdopodobnie ze względu na narzut synchronizacji i inne koszty zarządzania wątkami.

Wnioski te sugerują, że chociaż zwiększanie liczby wątków i wyłączanie mutexów może skracać czas obliczeń, to ich optymalny wybór zależy od potrzeb programu — szczególnie jeśli istotna jest spójność danych i stabilność działania.

4. Opis programu

Program obliczanie_calki.c został stworzony do obliczania całki funkcji sinus w przedziale od 0 do π, przy wykorzystaniu metody trapezów. W programie wykonuje się obliczenia sekwencyjne oraz równoległe, wykorzystując dwa sposoby dekompozycji wątków:

- Zrównoleglenie pętli: Każdy wątek oblicza częściową wartość całki dla przypisanego zestawu trapezów.
- Dekompozycja obszaru: Całkowany przedział dzieli się na mniejsze odcinki, z których każdy przypisany jest do oddzielnego wątku.

Wzorzec zrównoleglenia pętli:

- Funkcja calka_zrownoleglenie_petli realizuje ten wzorzec poprzez tworzenie określonej liczby wątków przez główny wątek. Każdy wątek wykonuje fragment iteracji pętli.
- Każdy wątek otrzymuje zakres indeksów na podstawie wyliczonej wartości: int j = ceil((float)(N/l_w));
- Następnie każdy wątek sumuje wartości funkcji w swoim przypisanym zakresie indeksów:

```
for (int i = 0; i < l_w; i++) {
  calka_global += tab_calka_global[i];
}
```

Koszt związany z użyciem mutexa i alternatywy bez mutexa:

- Wersja z mutexem generuje dodatkowe opóźnienia, ponieważ wymaga blokowania i odblokowywania dostępu do wspólnej zmiennej sumy w każdym wątku. W przypadku dużych zadań (np. przy dużej liczbie trapezów), te opóźnienia stają się mało istotne względem zysków z równoległego przetwarzania.

- Wersja bez mutexa, wykorzystująca tablicę global_array_of_local_sums, omija ten narzut. W tym podejściu wątki zapisują swoje sumy lokalne do oddzielnych komórek tablicy, a ich sumowanie następuje na końcu, w głównym wątku.

5. Wykonanie zadania nr.2
    - dekompozycja_petli.c

```c
#include<stdio.h>
#include<pthread.h>
#include<math.h>
#include<stdlib.h>

double funkcja ( double x );

double calka_zrownoleglenie_petli(double a, double b, double dx, int
l_w);

static int l_w_global=0;

static double calka_global=0.0;
static double *tab_calka_global;
static double a_global;
static double b_global;
static double dx_global;
static int N_global;

void* calka_fragment_petli_w(void* arg_wsk);
pthread_mutex_t mutex;
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;

  a_global = a;
  b_global = b;
  dx_global = dx_adjust;
  N_global = N;
  l_w_global = l_w;

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



  pthread_mutex_init(&mutex, NULL);
  int index [l_w];
  for (int i=0; i < l_w; i++){
    index [i] = i;
  }

  tab_calka_global = (double *) malloc(l_w*sizeof(double));
```

```
  pthread_t watki [l_w];
  for (int i=0; i<l_w ; i++){
    pthread_create(&watki[i], NULL, calka_fragment_petli_w,
(void*)(&index[i]));
  }

  for (int i=0; i<l_w ; i++){
    pthread_join(watki[i], NULL);
  }

    for (int i=0; i < l_w; i++){
    calka_global += tab_calka_global[i];
    }

  return(calka_global);
}

void* calka_fragment_petli_w(void* arg_wsk){

  int my_id =*((int*)arg_wsk);

  double a = a_global, b = b_global, dx = dx_global;
  int N = N_global, l_w = l_w_global;


  // dekompozycja cykliczna
  int my_start = my_id; //zaczynamy od nr watku
  int my_end = N; //ilosc danych / liczba watkow
  int my_stride = l_w;

  int j = ceil((float)(N/l_w));

  if (j*l_w > N) {
    printf("Error!");
  }
  // dekompozycja blokowa
  // int my_start = j * my_id;
  // int my_end = j * (my_id + 1);

  // if(my_id == l_w - 1){
  //  my_end = N;
  // }

  // int my_stride = 0;

  int i;
  double calka = 0.0;
```

```
  for(i=my_start; i<my_end; i+=my_stride+1){

    double x1 = a + i*dx;
    calka += 0.5*dx*(funkcja(x1)+funkcja(x1+dx));
  }
  // pthread_mutex_lock(&mutex);
  // calka_global += calka;
  // pthread_mutex_unlock(&mutex);
  tab_calka_global [my_id] = calka;

}
```

6. Analiza wyników
    - Wariant sekwencyjny obliczeń całki
      Wykonano sekwencyjny wariant obliczeń, w którym liczba trapezów (N) i wysokość (dx) zostały dopasowane do podziału przedziału [0, PI] na N trapezów. Wynik z obliczeń sekwencyjnych wyniósł:

      Wynik obliczonej całki: 1.570796326794897
      Czas wykonania: 0.000214 sekundy.

    - Wariant równoległy – zrównoleglenie pętli (cykliczne)
      Dla wariantu zrównoleglenia pętli, program utworzył 8 wątków, które równocześnie realizowały iteracje pętli z rozłożeniem iteracji w sposób cykliczny.
      Wyniki:

      Wynik obliczonej całki: 1.570796326794897 (wynik zbliżony do sekwencyjnego).
      Czas wykonania: 0.00320 sekundy.

    - Wariant równoległy – zrównoleglenie pętli (blokowe)
      Dla dekompozycji blokowej, zmieniono sposób dystrybucji iteracji, gdzie każdy wątek otrzymał bloki kolejnych iteracji pętli. Wynik tej metody był zgodny z wynikiem obliczeń sekwencyjnych, lecz różnił się nieznacznie czasem wykonania.
      Wyniki:

Wynik obliczonej całki: 1.570796326794897.
Czas wykonania: 0.00320 sekundy
naliza wyników i wnioski

7. Wnioski

Wyniki wariantu sekwencyjnego i równoległego ze zrównolegleniem pętli (cyklicznie i blokowo) były niemal identyczne, co potwierdza poprawność zrównoleglenia pętli.
Czas wykonania obliczeń w wariancie sekwencyjnym był najkrótszy (0.000214 s), jednak zrównoleglenie pętli (czas około 0.00320 s) pozwala na skalowanie obliczeń przy większej liczbie trapezów. Wykorzystanie zrównoleglenia pętli zwiększa czas wykonania w przypadku małych wartości N, co wynika z narzutu na tworzenie i synchronizację wątków.

Wynik wersji równoległej jest niemal identyczny z wynikiem wersji sekwencyjnej, ponieważ obie realizują te same obliczenia, lecz w wersji równoległej są one rozdzielane pomiędzy różne wątki. Każdy wątek przetwarza fragment danych i oblicza własną sumę lokalną, którą następnie dodaje do sumy globalnej. Z kolei w wersji sekwencyjnej to samo obliczenie jest wykonywane w ramach jednego wątku.

Jednak mimo uzyskania identycznych wyników, wersja równoległa może mieć pewne wady:

- Wersja z mutexem generuje dodatkowe opóźnienie związane z koniecznością blokowania i odblokowywania zasobów, co może obniżać efektywność, zwłaszcza przy mniejszych zadaniach

- Tworzenie, zarządzanie i synchronizacja wątków powodują dodatkowy czas wykonania, szczególnie przy zadaniach o małej skali, gdzie zysk z równoleglenia jest niewielki

- W wersji bez mutexów może występować zjawisko false sharing, gdy wątki zapisują do sąsiadujących komórek pamięci, co może powodować opóźnienia z powodu nieefektywnego dostępu do pamięci cache

- Dla dużych zadań zrównoleglenie przynosi zauważalny wzrost wydajności, jednak przy mniejszych problemach narzut może spowodować, że czas wykonania będzie dłuższy niż w przypadku wersji sekwencyjnej