Programowanie równoległe	Kierunek: Informatyka Techniczna	Grupa: 9
Imię i nazwisko: Karolina Starzec	Temat: LAB 4 I 5	Termin oddania: 13.11.2024

LAB 4

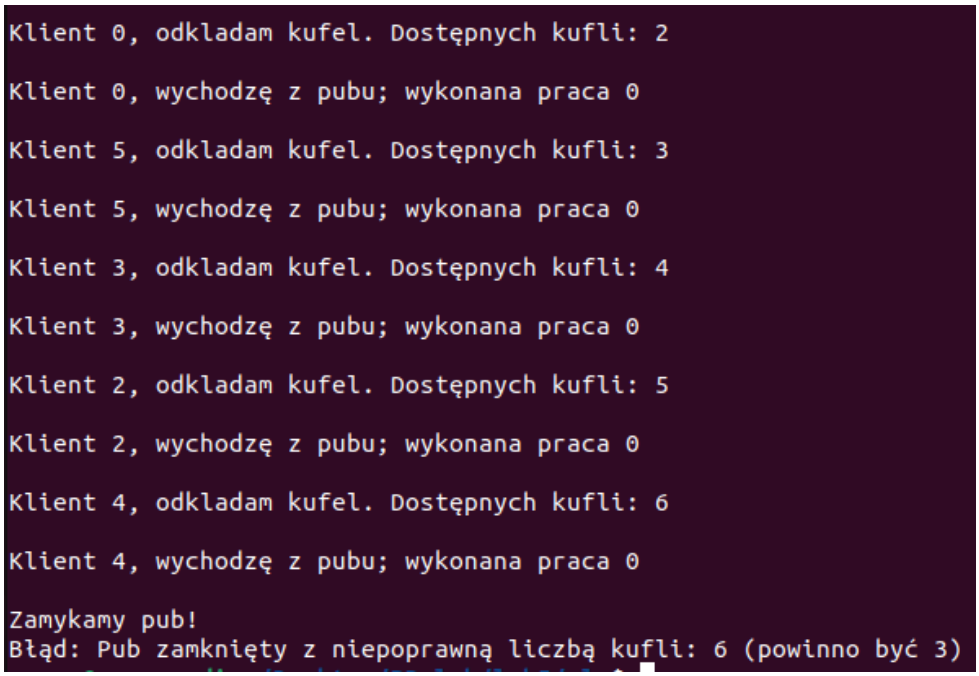
1. Cel

Opanowanie umiejętności pisania programów z synchronizacją wątków.

2. Realizacja zajęć

Realizując zajęcia skopiowałam i rozpakowałam potrzebne pliki ze strony przedmiotu oraz umieściłam je w odpowiednim podkatalogu. Następnie została przygotowana symulacja na podstawie pobranego kodu. Pierwsza wersja, jaką mieliśmy sprawdzić nie miała synchronizacji, przez co miały pojawić się potencjalne błędy do późniejszego rozwiązania.

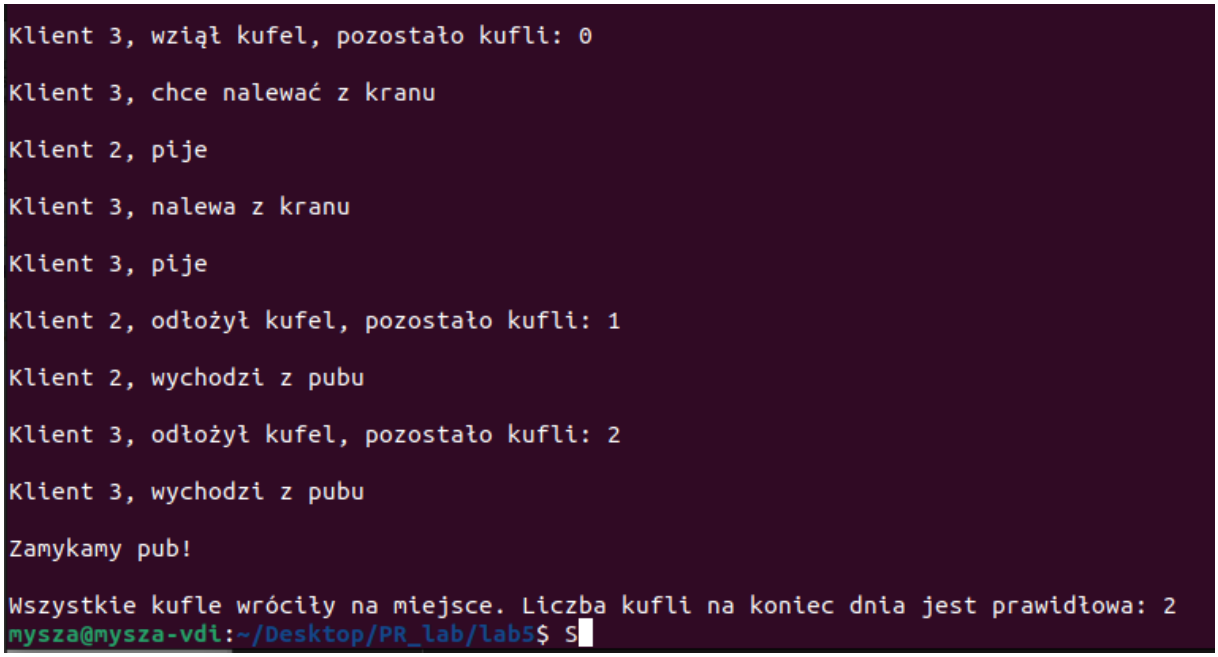
```
//sprawdzenie, czy liczba kufli na końcu jest taka sama jak na początku
  if (l_kf == initial_mug_count) {
     printf("Pub zamknięty z poprawną liczbą kufli: %d\n", l_kf);
  } else {
     printf("Błąd: Pub zamknięty z niepoprawną liczbą kufli: %d (powinno być %d)\n", l_kf,
initial_mug_count);
  }

//sprawdzenie, czy są jeszcze dostępne kufle

if (l_kf > 0) {

        l_kf--;

        printf("\nKlient %d, pobrał kufel. Dostępnych kufli: %d\n", moj_id, l_kf);

     }
```



Aby uniknąć konfliktów w dostępie do kufli, w poprawionym kodzie wprowadziłam synchronizację poprzez dodanie mutexa kufel_mutex. Dodanie pthread_mutex_lock i pthread_mutex_unlock przed i po każdej operacji dostępu do l_kufli gwarantuje, że tylko jeden wątek w danym momencie może modyfikować liczbę kufli. Dzięki blokowaniu i odblokowywaniu kufel_mutex unika się sytuacji, w której dwa wątki jednocześnie zmniejszają lub zwiększają liczbę dostępnych kufli, co w kodzie początkowym mogło prowadzić do niepoprawnych wyników.

```
// Pobranie kufla - synchronizacja za pomocą mutexa
pthread_mutex_lock(&kufel_mutex); // Blokowanie mutexa przed dostępem do kufli
if (l_kufli > 0) {
   l_kufli--; // Pobranie kufla
   printf("\\nKlient %d, wziął kufel, pozostało kufli: %d\\n", moj_id, l_kufli);
}
pthread_mutex_unlock(&kufel_mutex); // Odblokowanie mutexa po pobraniu kufla
```

```
// Oddanie kufla - synchronizacja za pomocą mutexa
pthread_mutex_lock(&kufel_mutex); // Blokowanie mutexa przed zwrotem kufla
l_kufli++; // Zwrócenie kufla
printf("\\nKlient %d, odłożył kufel, pozostało kufli: %d\\n", moj_id, l_kufli);
pthread_mutex_unlock(&kufel_mutex); // Odblokowanie mutexa po zwrocie kufla
```



Jaka najprostsza reprezentacja pozwala na rozwiązanie problemu bezpiecznego korzystania z kufli w pubie w przypadku liczby kufli większej od liczby klientów (jeden kufel jest posiadany tylko przez jednego klienta)?

Najprostszą reprezentacją, która pozwala na rozwiązanie problemu bezpiecznego korzystania z kufli w pubie, kiedy liczba kufli jest większa od liczby klientów, jest licznik dostępnych kufli kontrolowany przez jeden mutex. Ponieważ liczba kufli przekracza liczbę klientów, zawsze będzie dostępny przynajmniej jeden kufel dla każdego klienta. To upraszcza synchronizację, ponieważ klienci mogą swobodnie pobierać kufle, dopóki licznik kufli jest dodatni.

3. Wnioski

Jak wygląda rozwiązanie problemu bezpiecznego korzystania z kufli?
Rozwiązanie polega na użyciu licznika dostępnych kufli oraz mutexa, który zabezpiecza operacje pobierania i zwracania kufli. Klient blokuje mutex, sprawdza dostępność kufli, pobiera kufel lub czeka, a następnie odblokowuje mutex.

Jak połączyć je ze sprawdzeniem dostępności kufli?
Sprawdzenie dostępności odbywa się po zablokowaniu mutexa, gdzie klient weryfikuje, czy liczba dostępnych kufli jest większa od zera. Jeśli tak, pobiera kufel i zmniejsza licznik, jeśli nie – odblokowuje mutex i ewentualnie czeka.

Jaka jest wada rozwiązania z wykorzystaniem tylko mutexów?
Wada polega na tym, że brak kufli zmusza klienta do aktywnego sprawdzania ich dostępności, co prowadzi do nieefektywnego wykorzystania procesora. Bez innej mechaniki synchronizacji program marnuje zasoby procesora na nieustanne sprawdzanie stanu kufli w oczekiwaniu.

LAB 5

1. Cel

Nabycie umiejętności tworzenia i implementacji programów równoległych w środowisku Pthreads.

2. Realizacja zajęć

Realizując zajęcia skopiowałam i rozpakowałam potrzebne pliki ze strony przedmiotu oraz umieściłam je w odpowiednim podkatalogu. Następnie przeanalizowałam pierwszy plik pthreads_suma.c i przeprowadziłam testy.

Testy dla rozmiaru tablicy 100000000:

Sekwencyjne [s]	Wielowątkowe - mutex [s]	Wielowątkowe – tablica [s]
0.089715	0.048062	0.046578
0.089816	0.048022	0.047144
0.087375	0.047144	0.046657

Testy dla rozmiaru tablicy 1000000:

Sekwencyjne [s]	Wielowątkowe - mutex [s]	Wielowątkowe – tablica [s]
0.000865	0.000927	0.000669
0.001046	0.001056	0.000674
0.001116	0.000960	0.000669

Testy dla rozmiaru tablicy 100000:

Sekwencyjne [s]	Wielowątkowe - mutex [s]	Wielowątkowe – tablica [s]
0.000088	0.000467	0.000189
0.000087	0.000485	0.000171
0.000107	0.000435	0.000175

Testy dla rozmiaru tablicy 10000:

Sekwencyjne [s]	Wielowątkowe - mutex [s]	Wielowątkowe – tablica [s]
0.000008	0.000573	0.000170
0.000011	0.000390	0.000150
0.000010	0.000441	0.000152

Dla największego rozmiaru tablicy (100,000,000) obliczenia wielowątkowe zarówno z mutexem, jak i z tablicą lokalnych sum są wyraźnie szybsze niż obliczenia sekwencyjne, przy czym metoda z tablicą jest minimalnie szybsza od tej z mutexem. W przypadku tablicy o rozmiarze 1,000,000 różnice między metodami są mniejsze, ale metoda z tablicą nadal przewyższa czasowo mutex. Dla mniejszych rozmiarów (100,000 i 10,000) obliczenia wielowątkowe z tablicą są nadal szybsze niż te z mutexem, jednak różnica jest mniejsza, a przy najniższych rozmiarach tablic (10,000) koszty zarządzania wątkami przewyższają zyski z równoległego przetwarzania. Ogólnie metoda z tablicą lokalnych sum okazuje się bardziej efektywna niż metoda z mutexem, zwłaszcza przy dużych rozmiarach danych.

Jak wygląda prosty wzorzec zrównoleglenia pętli?

```
void *suma_w_no_mutex(void *arg_wsk) {
  int i, j, moj_id;
  moj_id = *( (int *) arg_wsk );

  double tmp = 0.0;
  j = ceil((float)ROZMIAR / LICZBA_W); // liczba elementów na wątek
  for (i = j * moj_id; i < j * (moj_id + 1) && i < ROZMIAR; i++) {
    tmp += tab[i];
  }
  global_array_of_local_sums[moj_id] = tmp;
  pthread_exit(NULL); }
```

W tym przykładzie każdy wątek oblicza sumę części tablicy tab i zapisuje wynik w global_array_of_local_sums. Każdy wątek otrzymuje unikalny identyfikator moj_id, który decyduje o tym, które indeksy tablicy będą przetwarzane przez dany wątek.

Jaki można zaobserwować narzut związany z wykonaniem równoległym – w wersji z mutex'em i bez mutex'a?

Narzut w wersji z mutexem wynika z konieczności synchronizacji dostępu do wspólnej zmiennej, co spowalnia działanie wątków, zwłaszcza dla mniejszych zadań. W wersji bez mutexa, korzystającej z tablicy lokalnych sum, narzut jest mniejszy, ponieważ wątki nie czekają na dostęp do zasobów, ale konieczne jest dodatkowe sumowanie wyników lokalnych.

Kolejnym krokiem podczas tych zajęć było pobranie paczki pthreads_calka.tgz. Rozpakowałam pliki w odpowiednim folderze i przetestowałam wstępną wersję kodu dla różnych wartości dx. Wynik dla całki sin(x) w przedziale od 0 do PI, zbiegał się do wartości 2.0, przy czym jego dokładność rosła wraz z zmniejszaniem wartości dx.

Następnie zmodyfikowałam kod o funkcję realizującą zrównoleglenie pętli, która tworzy wątki, dzieli iteracje i sumuję wyniki lokalne.

```
double calka_zrownoleglenie_petli(double a, double b, double dx, int l_w) {
  int N = ceil((b - a) / dx); // Obliczenie liczby trapezów
  double dx_adjust = (b - a) / N; // Korekta dx do uzyskania dokładnych podprzedziałów
  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_calka, NULL);

  pthread_t threads[l_w];
  int thread_ids[l_w];

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

```
    // Oczekiwanie na zakończenie pracy wątków
    for (int i = 0; i < l_w; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_mutex_destroy(&mutex_calka);
    return calka_global;
}
```

Mieliśmy to przetestować dla wersji z dekompozycją cykliczną oraz blokową.

```
void* calka_fragment_petli_w(void* arg_wsk) {
    int my_id = *(int*)arg_wsk;

    // Ustawienia dla dekompozycji cyklicznej
    int my_start = my_id;
    int my_stride = l_w_global;


    // Deklaracja lokalnej zmiennej dla wyników danego wątku
    double calka_local = 0.0;


    // Pętla obliczająca wartości trapezów w wariancie cyklicznym
    for (int i = my_start; i < N_global; i += my_stride) {
        double x1 = a_global + i * dx_global;
        calka_local += 0.5 * dx_global * (funkcja(x1) + funkcja(x1 + dx_global));
    }


    // Dodawanie wyniku lokalnego do globalnej zmiennej całki
    pthread_mutex_lock(&mutex_calka);
    calka_global += calka_local;
    pthread_mutex_unlock(&mutex_calka);

    return NULL;
}
```

```
//Alternatywna dekompozycja blokowa
  void* calka_fragment_petli_w(void* arg_wsk) {
  int my_id = *(int*)arg_wsk;

  // Ustawienia dla dekompozycji blokowej
  int my_block_size = (N_global + l_w_global - 1) / l_w_global;
  int my_start = my_id * my_block_size;
  int my_end = (my_start + my_block_size > N_global) ? N_global : my_start +
my_block_size;

  double calka_local = 0.0;

  for (int i = my_start; i < my_end; i++) {
    double x1 = a_global + i * dx_global;
    calka_local += 0.5 * dx_global * (funkcja(x1) + funkcja(x1 + dx_global));
  }

  pthread_mutex_lock(&mutex_calka);
  calka_global += calka_local;
  pthread_mutex_unlock(&mutex_calka);

  return NULL;
}
```

3. Wnioski

Dlaczego wynik wersji równoległej według wzorca zrównoleglenia pętli jest (nieomal) identyczny z wynikiem wersji sekwencyjnej?

Wynik wersji równoległej przy zrównolegleniu pętli jest niemal identyczny z wynikiem wersji sekwencyjnej, ponieważ obie wersje wykonują te same operacje na tych samych danych, tylko z różnym podziałem pracy.

Jakie można zauważyć wady wersji ze zrównolegleniem pętli?

Wady wersji z równolegleniem pętli obejmują konieczność synchronizacji dostępu do wspólnej zmiennej wynikowej, co wprowadza narzut czasowy na użycie mutexa.