|  |  |  |
| --- | --- | --- |
| 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:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, Czcionka, zrzut ekranu, design

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznieKlient 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.

Przykład błędu w którym występuje rywalizacja o kufle

• 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**

Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie• 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.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

• 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:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Wyniki dla ROZMIAR = 100000000 i LICZBA\_W = 2:

Obraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

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

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Wysokość trapezu = 0.002, ilość wątków = 2

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Wysokość trapezu = 0.0002, ilość wątków = 2

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

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.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

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.