Imię i Nazwisko Patryk Twardosz	Kierunek Informatyka Techniczna	Rok studiów i grupa I rok, Gr. 9
Data zajęć 28.10.2024r.	Numer i temat sprawozdania Lab 4 i 5 – Pthread	

Lab4:

Cel:

- Nauczyć się zarządzania dostępem do zasobów współdzielonych za pomocą mutexów.

fork() - Tworzy nowy proces jako kopię procesu rodzica.

clone() - Bardziej elastyczna funkcja, umożliwiająca większą kontrolę nad współdzielonymi zasobami.

Zadanie:

1. Przygotowanie projektu
2. Analiza problemu dostępu do wspólnej zmiennej (Kufli) przez wiele wątków (Klientów).

   - pub_sym_0.c: Implementacja wydawania kufli (sprawdzanie ilości kufli na konie c dnia):

```c
int dostepne_kufle;

void wybierzKufel() {
  dostepne_kufle--;
}

void oddajKufel(int kufel) {
  dostepne_kufle++;
}

int main(void) {
  /* Funkcjonowanie pub'u */

  // Zamykanie pub'u
  int roznica_kufli = dostepne_kufle - l_kf;
  if (roznica_kufli != 0)
    fprintf(stderr, "\nRóznica w liczbie kufili wynosi: %d\n", roznica_kufli);
  printf("\nZamykamy pub!\n");
}
```

```
Róznica w liczbie kufili wynosi: 1

Zamykamy pub!
```

```
Róznica w liczbie kufili wynosi: -3

Zamykamy pub!
```

3. pub_sym_1.c: Naprawa korupcji danych (nadmiarowych/brakujących kufli na koniec dnia):

```c
pthread_mutex_t kufel_mutex;
int dostepne_kufle;

void wybierzKufel() {
  pthread_mutex_lock(&kufel_mutex);
  if (dostepne_kufle < 0)
    fprintf(cerr, "Brakuje kufli.\n");
  dostepne_kufle--;
  pthread_mutex_unlock(&kufel_mutex);
}

void oddajKufel(int kufel) {
  pthread_mutex_lock(&kufel_mutex);
  dostepne_kufle++;
  pthread_mutex_unlock(&kufel_mutex);
}
```

Problem z pojawianiem się i znikaniem kufli został rozwiązany, natomiast dalej istnieje problem, gdy liczba klientów jest większa od liczby kufli. Doprowadza to do sytuacji, gdzie liczba dostępnych kufli jest ujemna.

```
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
Brakuje kufli.
```

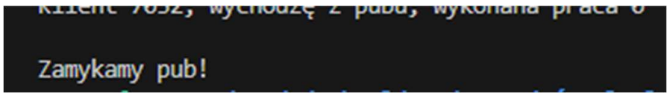
4. pub_sym_2.c: Implementacja aktywnego czekania.

```c
pthread_mutex_t kufel_mutex;
int dostepne_kufle;


int wybierzKufel() {
  do {
    pthread_mutex_lock(&kufel_mutex);
    if (dostepne_kufle > 0) {
      dostepne_kufle--;
      pthread_mutex_unlock(&kufel_mutex);
      return 0;
    }

    pthread_mutex_unlock(&kufel_mutex);
    do_something_else_or_nothing();
  } while (true);
}

void oddajKufel(int kufel) {
  pthread_mutex_lock(&kufel_mutex);
  dostepne_kufle++;
  pthread_mutex_unlock(&kufel_mutex);
}
```



Pub funkcjonuje i zamyka się poprawnie, bez żadnych nadmiarowych, czy brakujących kufli.

Natomiast dalej nie jest to najlepsze rozwiązanie, zważając na to, że każdy wątek (Klient) musi blokować dostęp do danych, po to tylko żeby sprawdzić czy są dostępne kufle.

5. pub_sym_2.c: Implementacja pasywnego czekania (wykorzystanie trylock)

```
int wybierzKufel(long int* work) {
  do {
    if (pthread_mutex_trylock(&kufel_mutex) == 0) {
      if (dostepne_kufle > 0) {
        dostepne_kufle--;
        pthread_mutex_unlock(&kufel_mutex);
        return 0;
      }
      pthread_mutex_unlock(&kufel_mutex);
    }

    do_something_else_or_nothing(work);
  } while (true);
}
```

Teraz wątki (Klienci) w trakcie braku dostępnych kufli wykonują inną prace, nie blokując dostępu do zasobów współdzielonych.

Wnioski:

- **Mutexy** pozwalają na synchronizację dostępu do zasobów współdzielonych między wątkami, eliminując problemy wynikające z wyścigów danych (*race condition*). Dzięki mutexom tylko jeden wątek na raz może uzyskać dostęp do zasobu, co zapewnia spójność danych.
- **Lock** (pthread_mutex_lock()) służy do zablokowania mutexu przed wejściem do sekcji krytycznej. Gdy wątek uzyska blokadę, inne wątki próbujące zablokować ten sam mutex zostają zatrzymane do momentu jego zwolnienia (pthread_mutex_unlock()). Mechanizm ten jest skuteczny, ale może prowadzić do zakleszczeń, jeśli wątki wzajemnie blokują dostęp do zasobów.
- **Trylock** (pthread_mutex_trylock()) umożliwia wątkowi próbę zablokowania mutexu bez czekania, co jest przydatne w sytuacjach, gdzie blokowanie wątku mogłoby pogorszyć wydajność. Jeśli mutex jest już zablokowany, trylock natychmiast zwraca kontrolę z odpowiednim komunikatem, co pozwala na efektywniejsze zarządzanie czasem i zasobami. Ten mechanizm jest przydatny do implementacji nieblokujących struktur danych, jednak jego stosowanie wymaga dodatkowej logiki obsługi.

Odpowiedzi na pytania:

- Najprostszą reprezentacją do bezpiecznego korzystania z kufli przy większej liczbie kufli niż klientów jest **użycie semafora licznikowego**. Każdy kufel jest traktowany jako zasób w puli, a semafor zlicza liczbę dostępnych kufli. Kiedy klient chce wziąć kufel, wykonuje operację P (wait) na semaforze – jeśli kufel jest dostępny, klient może go użyć. Po skończeniu wykonuje V (signal), zwalniając kufel dla innych. Takie rozwiązanie zapewnia prostotę i eliminuje zakleszczenia, gwarantując dostęp tylko wtedy, gdy kufel jest wolny.

- Rozwiązaniem problemu bezpiecznego korzystania z kufli jest użycie semaforu licznikowego do monitorowania dostępności kufli oraz mutexu do synchronizacji dostępu. Klient, chcąc wziąć kufel, wykonuje P (wait) na semaforze – jeśli kufel jest dostępny, może go zabrać, a po skończeniu wykonuje V (signal), zwalniając kufel. Wadą rozwiązania opartego wyłącznie na mutexach jest konieczność aktywnego czekania, co prowadzi do marnowania zasobów procesora, jeśli program nie ma innych operacji do wykonania w czasie oczekiwania.
- Schemat działania pojedynczego wątku, który chce bezpiecznie korzystać z kufli, można zaimplementować za pomocą semaforu licznikowego oraz mutexu. Semafor licznikowy pozwala uniknąć bezproduktywnego czekania, a mutex synchronizuje dostęp do sekcji krytycznej, gwarantując bezpieczeństwo operacji na współdzielonych zasobach.
- Użycie funkcji trylock() w kontekście symulacji oczekiwania na picie piwa nie zwiększa bezpośrednio ilości wykonanej pracy, ponieważ trylock() działa na zasadzie nieblokującego żądania. Jeśli nie uda się zablokować zasobu (np. kufla), wątek może kontynuować inne zadania lub spróbować ponownie później. Wynik działania tego mechanizmu zależy od liczby klientów oraz proporcji liczby klientów do liczby dostępnych kufli. Większa liczba klientów w stosunku do liczby kufli sprawia, że wątki częściej będą musiały czekać na dostęp do zasobu, co może prowadzić do mniejszej efektywności, ale nie zwiększa to samej ilości wykonanej pracy, a jedynie czas oczekiwania na dostęp do kufla.

Lab5:

Cel:

- Nabycie umiejętności tworzenia i implementacji programów równoległych w środowisku Pthreads

Zadanie:

1. Przygotowanie projektu
2. Uzupełnienie kodu programu *dekompozycja_petli.c*: Implementacja dekompozycji cyklicznej i blokowej

```
double calka_zrownoleglenie_petli(double a, double b, double dx, int l_w){
  // Przypisanie zmiennych globalnych
  calka_global = 0;
  l_w_global = l_w;
  a_global = a;
  b_global = b;
  dx_global = dx_adjust;
  N_global = N;
  // tworzenie wątków i mutex'u
  pthread_mutex_init(&mtx, NULL);
  for (int i = 0; i < l_w; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, calka_fragment_petli_w, &ids[i]);
  }  …
}

void* calka_fragment_petli_w(void* arg_wsk){
// Obliczanie parametrów pętli
#define CYCLIC
#ifdef CYCLIC
  // dekompozycja cykliczna
  int my_start = my_id;
  int my_end = N;
  int my_stride = l_w;
#else
  // dekompozycja blokowa
  int my_start = j * my_id;
  int my_end = j * (my_id + 1);
  int my_stride = 1;
#endif

  /* Obliczanie całki */

  // sumowanie całki
  pthread_mutex_lock(&mtx);
  calka_global += calka;
  pthread_mutex_unlock(&mtx);
}
```

3. dekompozycja_obszaru.c: Implementacja dekompozycji obszaru

```c
typedef struct {
  pthread_t thread;
  int id;
  double a, b, dx;

  double calka;
} ThreadData;

double calka_dekompozycja_obszaru(double a, double b, double dx, int l_w){
  // tworzenie struktur danych do obsługi wielowątkowości
  ThreadData* data = malloc(l_w * sizeof(ThreadData));

  double d = (b - a) / l_w;

  // tworzenie wątków
  for (int i = 0; i < l_w; i++) {
    data[i].id = i;
    data[i].a = a + d * i;
    data[i].b = a + d * (i + 1);
    data[i].dx = dx;

    pthread_create(&data[i].thread, NULL, calka_podobszar_w, &data[i]);
  }

  // oczekiwanie na zakończenie pracy wątków
  for (int i = 0; i < l_w; i++) {
    pthread_join(data[i].thread, NULL);
    calka_suma_local += data[i].calka;
  }
}
void* calka_podobszar_w(void* arg_wsk){

  // rozpakowanie danych przesłanych do wątku
  ThreadData* data = arg_wsk;
  double a_local = data->a, b_local = data->b, dx = data->dx;
  int my_id = data->id;

  data->calka = calka_sekw(a_local, b_local, dx);
}
```

4. Implementacja dekompozycji obszaru z wykorzystaniem pthread_exit();

```c
double calka_dekompozycja_obszaru(double a, double b, double dx, int l_w){
  /* Wystko tak jak w poprzednim punkcie */

  // oczekiwanie na zakończenie pracy wątków
  for (int i = 0; i < l_w; i++) {
    double* result;
    pthread_join(data[i].thread, &result);
    calka_suma_local += *result;
    free(result);
  }
}

void* calka_podobszar_w(void* arg_wsk){
  // rozpakowanie danych przesłanych do wątku
  ThreadData* data = arg_wsk;
  double a_local = data->a, b_local = data->b, dx = data->dx;
  int my_id = data->id;
  double* calka = malloc(sizeof(double));
  *calka = calka_sekw(a_local, b_local, dx);
  pthread_exit(calka);
}
```

5. Optymalizacja funkcji *calka_sekw()*:

```c
double calka_sekw(double a, double b, double dx){
  int N = ceil((b-a)/dx);
  double dx_adjust = (b-a)/N;
  int i;
  double calka = 0.0;
  double prev = funkcja(a);
  double dx_half = 0.5 * dx_adjust;

  for(i=0; i<N; i++){
    double x1 = a + (i + 1)*dx_adjust;
    double curr = funkcja(x1+dx_adjust);

    calka += dx_half * (prev + curr);

    prev = curr;
  }

  return calka;
}
```

6. Wyniki czasowe:

Całka Sekwencyjna:

dx	Całka	Czas [s]	Czas [ms]
0,1	1,983959	0,000015	0,015
0,01	1,999834	0,000015	0,015
0,001	1,999998	0,000036	0,036
0,0001	2	0,000325	0,325
0,00001	2	0,002478	2,478
0,000001	2	0,025296	25,296
0,0000001	2	0,239644	239,644

Liczba Wątków	Czas Sek [s]	Czas Cykl [s]	Czas Blok [s]	Czas Obsz [s]
1	0.247512	0.460940	0.424638	0.242368
2	0.240883	0.241404	0.217185	0.133135
4	0.240241	0.134026	0.131374	0.075291

Wnioski:

- **Efektywność równoległości**: Wykorzystanie Pthreads do równoległego obliczania całki znacząco przyspiesza proces w porównaniu do wersji sekwencyjnej, zwłaszcza przy dużej liczbie przedziałów podziału. Każdy wątek może niezależnie wykonywać obliczenia na przypisanym zakresie, co pozwala lepiej wykorzystać zasoby procesora.
- **Wydajność i liczba wątków**: Wydajność obliczeń może różnić się w zależności od liczby wątków. Optymalna liczba wątków zależy od liczby dostępnych rdzeni procesora — zwiększenie liczby wątków powyżej liczby rdzeni może skutkować nadmiernym narzutem wynikającym z przełączania kontekstu.

Odpowiedzi na pytania:

- **Prosty wzorzec zrównoleglenia pętli** w obliczeniach przy użyciu wątków Pthreads polega na podzieleniu iteracji pętli na mniejsze bloki, które są przetwarzane równolegle przez różne wątki.
- Wynik wersji równoległej według wzorca zrównoleglenia pętli jest niemal identyczny z wynikiem wersji sekwencyjnej, ponieważ każdy wątek wykonuje dokładnie tę samą operację na danych, które są jedynie podzielone na różne zakresy. W efekcie sumaryczny wynik obliczeń wszystkich wątków jest taki sam, jak wynik uzyskany w wersji sekwencyjnej, przy założeniu, że nie występują błędy współbieżności i synchronizacja działa poprawnie.
- Dekompozycja w dziedzinie problemu dla obliczenia całki oznacza, że obszar całkowania (np. przedział na osi x) jest dzielony na mniejsze części, a każdy wątek oblicza całkę na przypisanym mu podprzedziale.