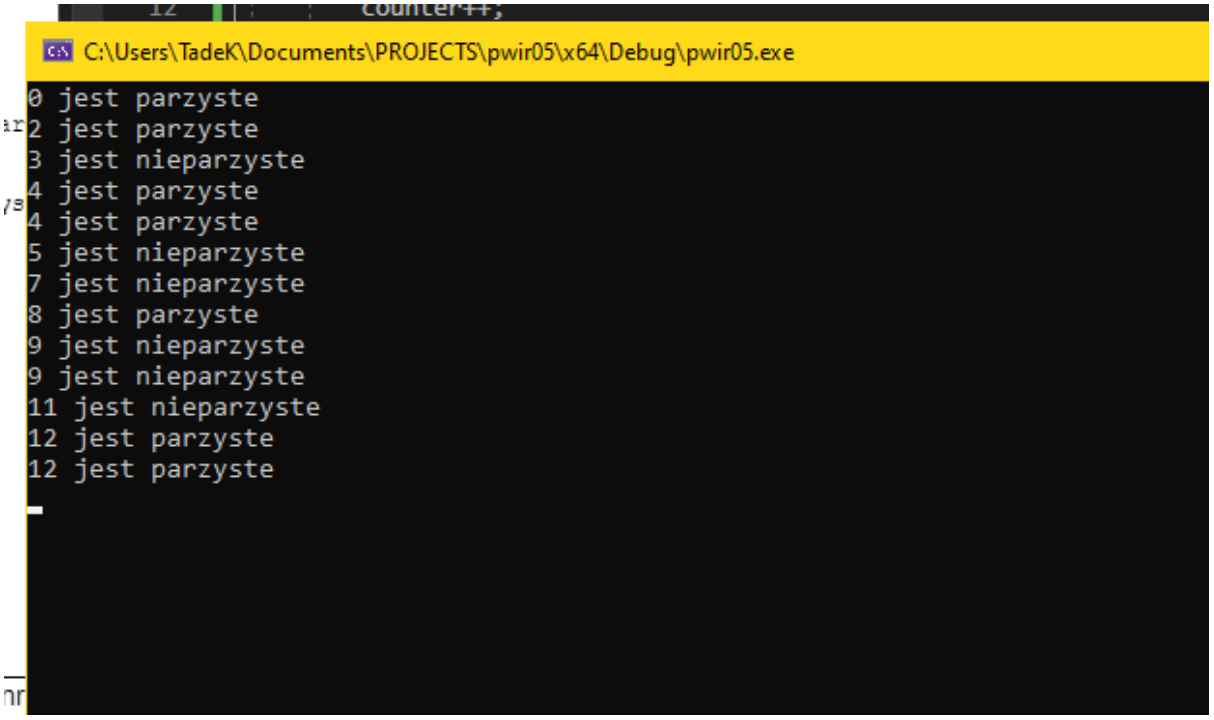
Akademia Nauk Stosowanych w Nowym Sączu Programowanie Współbieżne i Rozproszone			
Temat: Przykład z parzystością.			
Nazwisko i imię: Ciapała Tadeusz		**Ocena sprawozdania**	**Zaliczenie:**
Data wykonania ćwiczenia: 21.03.2023	**Grupa: P3**		

1) **KOD 7 i 8:**

Na początku zadanych do wykonania zadań należało zapoznać się z kodem numer 7 i 8. Przedstawiał on przykład ze sprawdzaniem parzystości danej liczby, używając do tego 2 wątków. Wątek nr 1 miał za zadanie inkrementację zmiennej globalnej, natomiast wątek nr 2 oceniał parzystość inkrementowanej zmiennej.

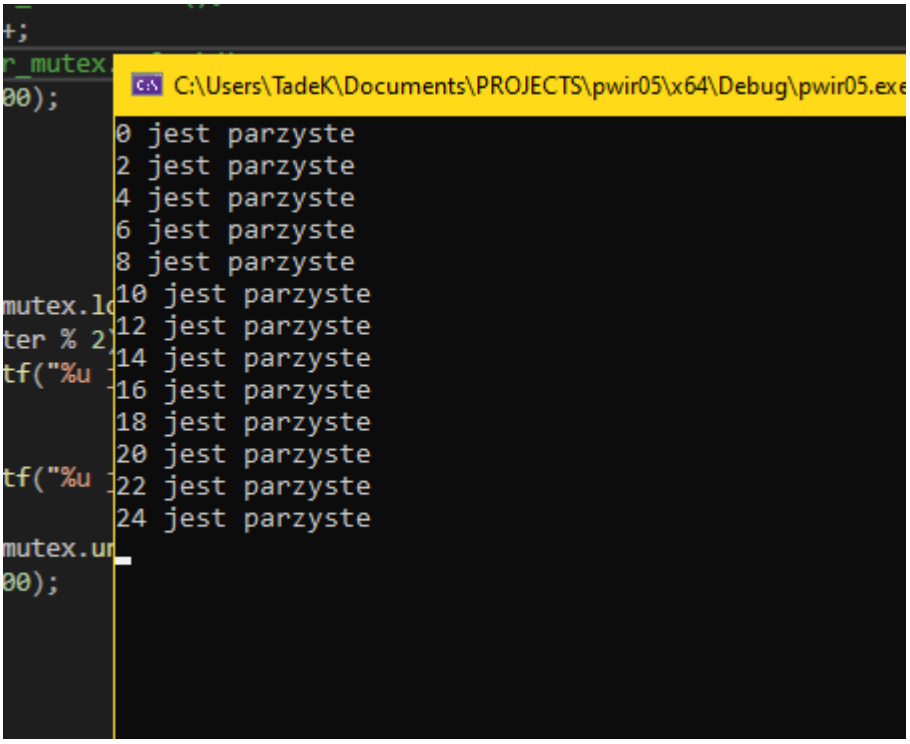
```
#include <thread>
#include <cstdio>
#include <windows.h>
#include <mutex>

std::mutex counter_mutex;
unsigned int counter = 0;

void increment(){
    for(;;){
        counter_mutex.lock();
        counter++;
        counter_mutex.unlock();
        Sleep(2000);
    }
}

void parity(){
    for(;;){
        counter_mutex.lock();
        if (counter % 2){
            printf("%u jest nieparzyste\r\n", counter);
        }
        else{
            printf("%u jest parzyste\r\n", counter);
        }
        counter_mutex.unlock();
        Sleep(2000);
    }
}

int main(){
    std::thread inc(increment);
    std::thread par(parity);

    inc.join();
    par.join();

    printf("Done\r\n");

    return 0;
}
```

Listing nr 1: dostarczy do zadań kod nr 7 i 8



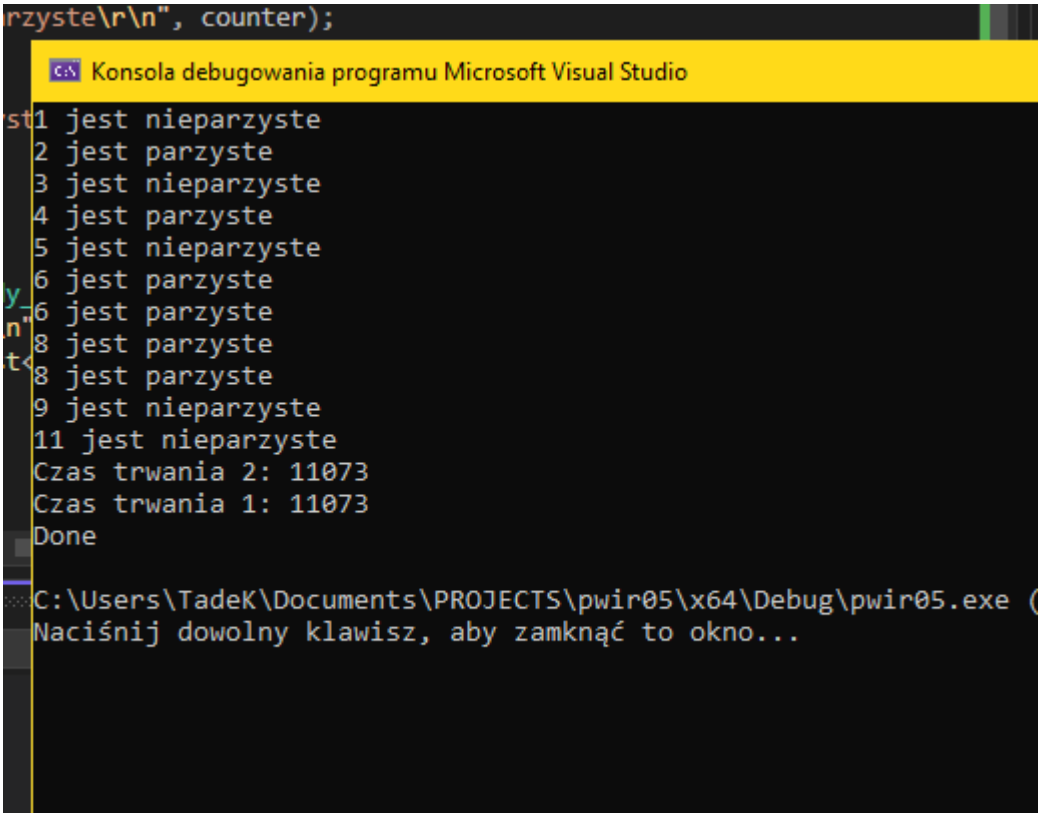
Zrzut nr 1: pierwsze uruchomienie kodu 7/8.

Pierwszą rzeczą którą widzimy jest wartość licznika, która wydaje się być błędnie obliczana. Kolejną rzeczą jest sama parzystość. Jeśli byśmy poczekali dłużej okazałoby się, że parzystość liczby nie zawsze jest dobrze wskazywana. Jest to spowodowane błędnym rozwiązaniem przedstawionego problemu (undefined behaviour) , dlatego też należy użyć biblioteki cpp - mutex:



Zrzut nr 2: kod 7/8 po usuniecie blokowania i odblokowania w jednym z wątków

W pierwszym zadaniu należało usunąć mechanizm blokady licznika w jednym z wątków. Po zakomentowaniu odpowiednich linii, zaobserwować można było zjawisko pokazane na zrzucie nr 2. Wypisywane były tylko liczby parzyste.



Zrzut nr 3: wykonanie zadania 2 do kodu 7 i 8

```
#pragma warning( disable : 4473 )
#include <thread>
#include <cstdio>
#include <windows.h>
#include <mutex>
#include <chrono>
#include <iostream>

std::mutex counter_mutex;
unsigned int counter = 0;

void increment() {
    auto start = std::chrono::steady_clock::now();
    for (;;) {
        if (counter == 11)
            break;
        counter_mutex.lock();
        counter++;
        counter_mutex.unlock();
        Sleep(1000);
    }
    auto end = std::chrono::steady_clock::now();
    printf("Czas trwania 1: %llu\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count()
    );
}
```

```
void parity() {
    auto start = std::chrono::steady_clock::now();
    for (;;) {
        if (counter == 11)
            break;
        counter_mutex.lock();
        if (counter % 2) {
            printf("%u jest nieparzyste\r\n", counter);
        }
        else {
            printf("%u jest parzyste\r\n", counter);
        }
        counter_mutex.unlock();
        Sleep(1000);
    }
    auto end = std::chrono::steady_clock::now();
    printf("Czas trwania 2: %llu\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count()
    );
}

int main() {
    std::thread inc(increment);
    std::thread par(parity);

    inc.join();
    par.join();

    printf("Done\r\n");

    return 0;
}
```

Listing nr 2: rozwiązanie do zadania nr 2 kodu 7 i 8

Zmierzone zostały dokładnie czasy wykonania pętli dla obu wątków. Ustawiłem punkt zakończenia pętli na moment w którym zmienna “counter” osiąga wartość 11. Na zrzucie nr 3 widzimy, że zmierzone czasy dla obu pętli były identyczne.

2) KOD 9:

Kod ten dotyczył zmiennych lokalnych w danym wątku. Przedstawiał on typ zmiennej thread_local, którego zastosowanie powodowało, że każdy z wątków otrzymywał kopię zmiennej “counter”.

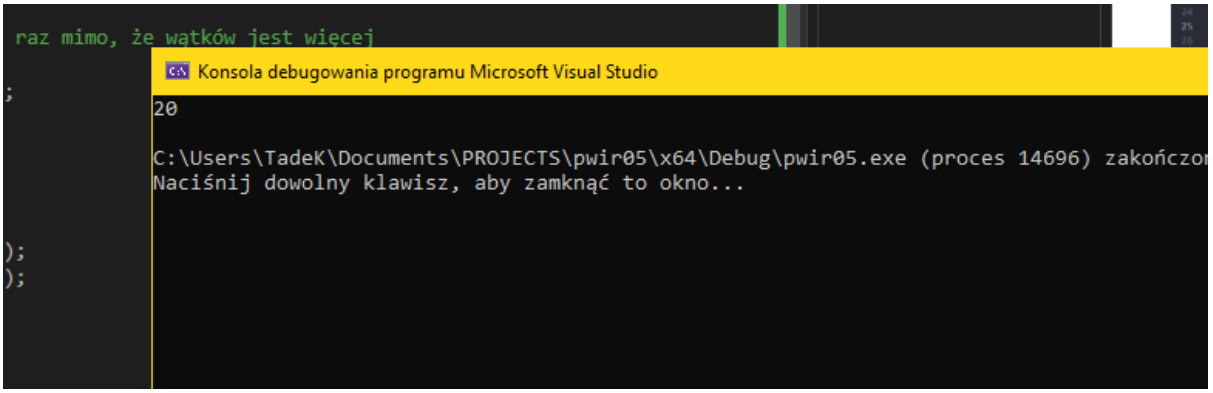
```
#include <thread>
#include <cstdio>
#include <windows.h>

unsigned int counter = 0;

void increment(int id){
    for(int i = 0;i<10;i++){
        counter++;
        Sleep(300);
    }

    //ten blok wykona się tylko raz mimo, że wątków jest więcej
    if(id == 1){
        printf("%u\n", counter);
    }
}
```

```
int main(){

    std::thread t1(increment,1);
    std::thread t2(increment,2);

    t1.join();
    t2.join();

    return 0;
}
```

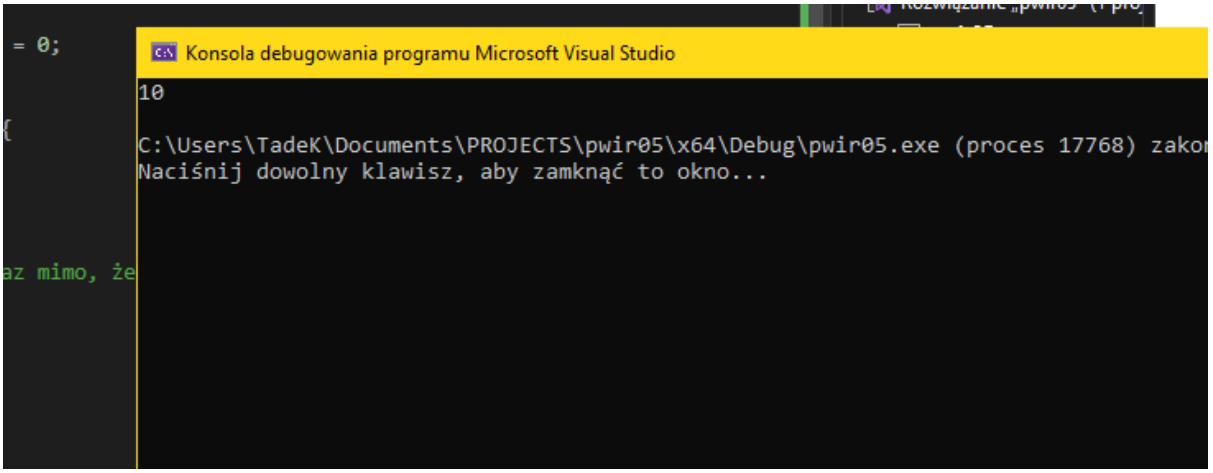
Listing nr 3: kod nr 9, wersja bez thread_local

```
#include <thread>
#include <cstdio>
#include <windows.h>

thread_local unsigned int counter = 0;

void increment(int id){
    for(int i = 0;i<10;i++){
        counter++;
        Sleep(300);
    }

    //ten blok wykona się tylko raz mimo, że wątków jest więcej
    if(id == 1){
        printf("%u\n", counter);
    }
}

int main(){

    std::thread t1(increment,1);
    std::thread t2(increment,2);

    t1.join();
    t2.join();

    return 0;
}
```

Listing nr 4: kod nr 9, wersja z thread_local



Zrzut nr 4: pierwsze uruchomienie pierwszej wersji kodu nr 9



Zrzut nr 5: pierwsze uruchomienie drugiej wersji kodu nr 9

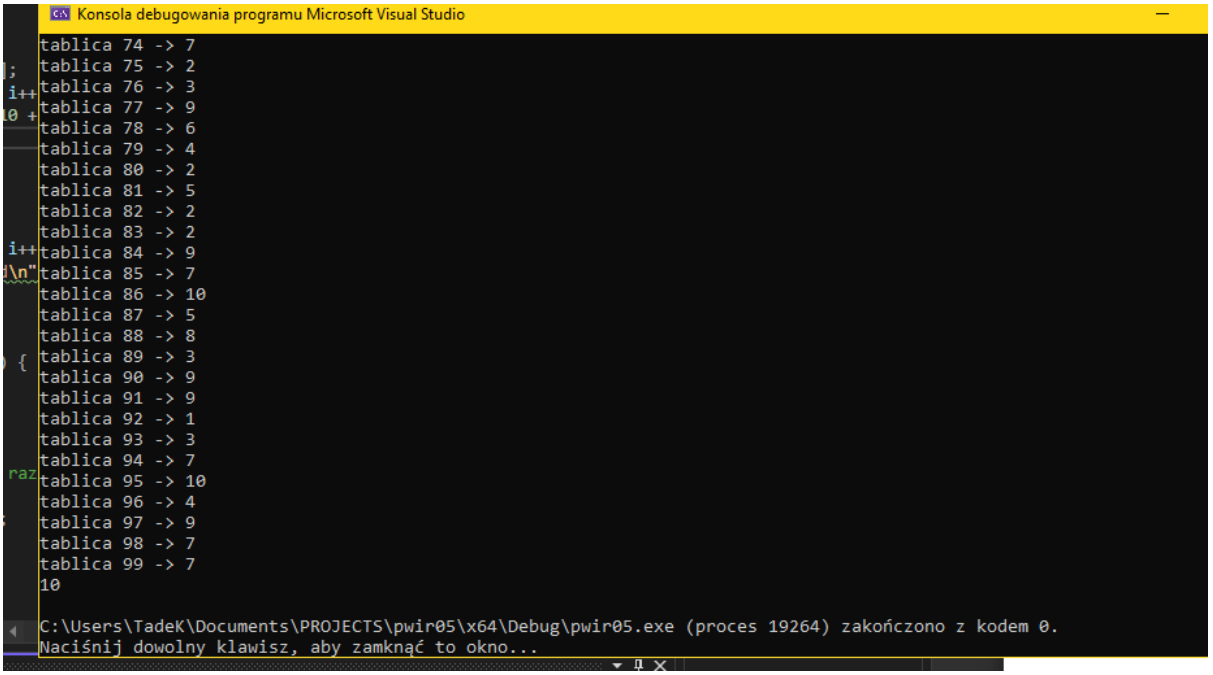
Jak widać wyniki różnią się, ponieważ w drugim przypadku jako licznik do wątku przekazywana jest kopia.

```
#include <thread>
#include <cstdio>
#include <windows.h>
#include <time.h>

thread_local unsigned int counter = 0;

int* generowanie()
{
    srand(time(NULL));
    int* my_array = new int[100];
    for (size_t i = 0; i < 100; i++)
        my_array[i] = rand() % 10 + 1;
    return my_array;
}

void wypisywanie(int* my_array)
{
    for (size_t i = 0; i < 100; i++)
        printf("tablica %d -> %d\n",i,my_array[i]);
}

void increment(int id) {
    for (int i = 0; i < 10; i++) {
        counter++;
        Sleep(300);
    }

    //ten blok wykona się tylko raz mimo, że wątków jest więcej
    if (id == 1) {
        printf("%u\n", counter);
    }
}

int main() {
    int* tab = generowanie();
    wypisywanie(tab);

    std::thread t1(increment, 1);
    std::thread t2(increment, 2);

    t1.join();
```

```
    t2.join();

    return 0;
}
```

Listing nr 5: rozwiązanie zadania nr 1

Na listingu nr 5 widać, żę do kodu zostały dodane dwie funkcje. Pierwsza z nich ma za zadanie zaalokować pamięć dla tablicy 100-elementowej oraz wypełnić ją losowymi wartościami od 1 do 10. Druga funkcja wypisuje jej zawartość.

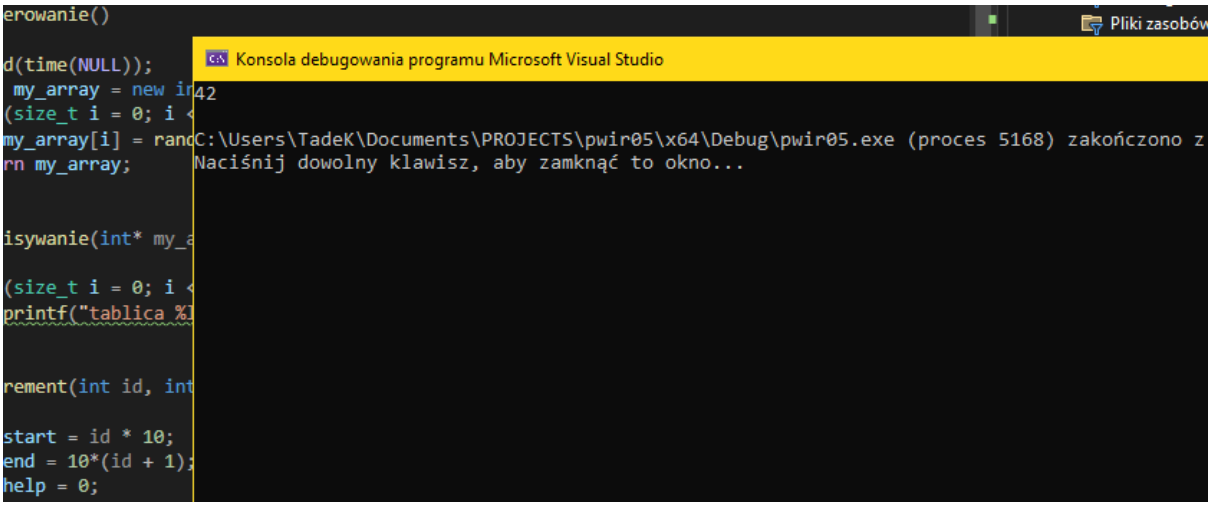


Zrzut nr 6: rozwiązanie zadania nr 1 do kodu nr 9

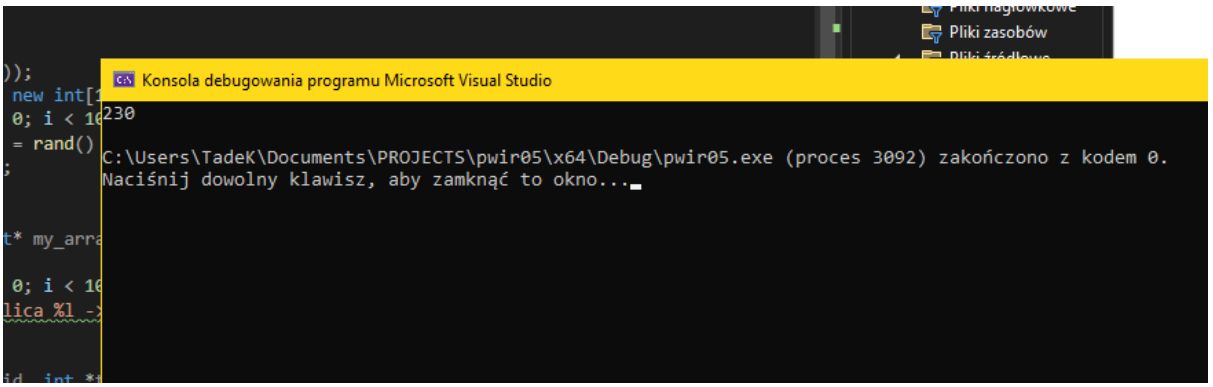
```
#include <thread>
#include <cstdio>
#include <windows.h>
#include <time.h>


thread_local unsigned int counter = 0;
//unsigned int counter = 0;


int* generowanie()
{
    srand(time(NULL));
    int* my_array = new int[100];
    for (size_t i = 0; i < 100; i++)
        my_array[i] = rand() % 10 + 1;
    return my_array;
}

void wypisywanie(int* my_array)
{
    for (size_t i = 0; i < 100; i++)
        printf("tablica %l -> %d\n",i,my_array[i]);
}

void increment(int id, int *tab)
```

```
{
    int start = id * 10;
    int end = 10*(id + 1);
    int help = 0;
    for (size_t i = start; i < end; i++)
    {
        help = help + tab[i];
        Sleep(100);
    }

    counter += help;
    //ten blok wykona się tylko raz mimo, że wątków jest więcej
    if (id == 1) {
        printf("%u\n", counter);
    }
}

int main() {
    int* tab = generowanie();
    //wypisywanie(tab);

    std::thread threads[10];
    for (size_t i = 0; i < 10; i++) {
        threads[i] = std::thread(increment, i, tab);
    }

    for (size_t i = 0; i < 10; i++) {
        threads[i].join();
    }

    delete[] tab;
    return 0;
}
```

Listing nr 6: rozwiązanie zadania nr 2



Zrzut nr 7: wykonanie programu dla zadania nr 2 - wersja z thread_local



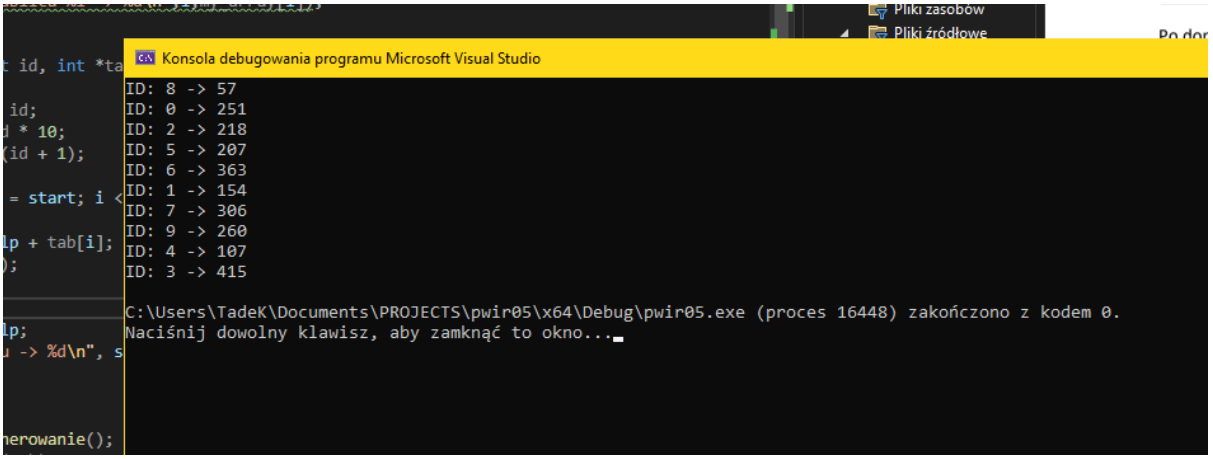
Zrzut nr 8: wykonanie programu dla zadania nr 2 - wersja bez thread_local

Na zrzutach nr 7 i 8 widzimy, że dwa różne typy zmiennych dają dwa różne wyniki. Dla rozwiązania thread_local, jak wiemy każdy wątek posiada swoją własną kopię, natomiast drugie rozwiązanie sumuje licznik globalnie.

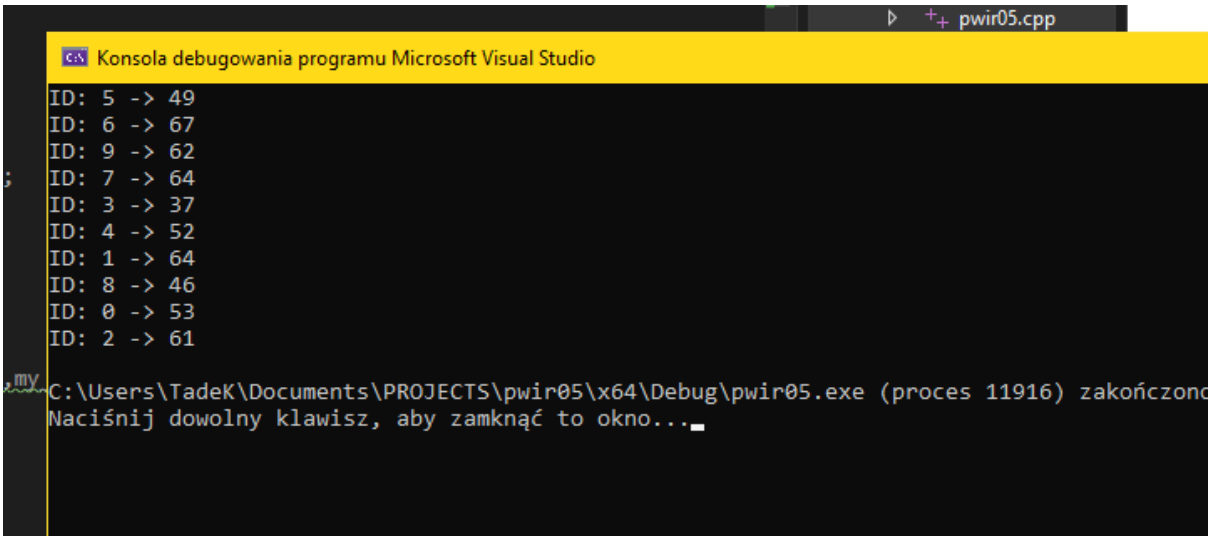
```
#include <thread>
#include <cstdio>
#include <windows.h>
#include <time.h>


//thread_local unsigned int counter = 0;
unsigned int counter = 0;


int* generowanie()
{
    srand(time(NULL));
    int* my_array = new int[100];
    for (size_t i = 0; i < 100; i++)
        my_array[i] = rand() % 10 + 1;
    return my_array;
}

void wypisywanie(int* my_array)
{
    for (size_t i = 0; i < 100; i++)
        printf("tablica %l -> %d\n",i,my_array[i]);
}

void increment(int id, int *tab)
{
    int save_id = id;
    int start = id * 10;
    int end = 10*(id + 1);
    int help = 0;
    for (size_t i = start; i < end; i++)
    {
        help = help + tab[i];
        Sleep(100);
    }

    counter += help;
    printf("ID: %u -> %d\n", save_id, counter);
}

int main() {
    int* tab = generowanie();
    //wypisywanie(tab);
```

```
    std::thread threads[10];
    for (size_t i = 0; i < 10; i++) {
        threads[i] = std::thread(increment, i, tab);
    }

    for (size_t i = 0; i < 10; i++) {
        threads[i].join();
    }

    delete[] tab;
    return 0;
}
```

Listing nr 7: rozwiązanie zadania nr 3



Zrzut nr 9: uruchomienie rozwiązania do zadania 3 - bez thread_local



Zrzut nr 10: uruchomienie rozwiązania do zadania 3 - wraz z użyciem thread_local

Teraz, gdy nasz program wypisuje id wątku oraz obliczane wartości, dokładniej możemy obserwować jak zachowują się dwa przedstawione podejścia do dzielenia się zmienną globalną.