PROGRAMOWANIE OBIEKTOWE – LABORATORIUM

*ĆWICZENIE 8 - WIELOWĄTKOWOŚĆ I SYNCHRONIZACJA*

---

## 1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z wzorcem obiektowym puli obiektów oraz utrwalanie praktycznej umiejętności tworzenia aplikacji działających wielowątkowo, a także stosowania mechanizmów synchronizacji danych współdzielonych miedzy wątkami.

## 2. Pojęcia

**Proces** – w systemie operacyjnym proces jest instancją pojedynczego, wykonywanego programu. System operacyjny przydziela procesowi zasoby. W ramach jednego procesu może działać wiele wątków.

**Wątek** - część programu wykonywana współbieżnie (ale z reguły nie równolegle), w obrębie jednego procesu. W procesie może pracować wiele wątków. Każdy z wątków może w czasie pojedynczego taktu procesora zrealizować pojedynczą instrukcję (zarezerwować czas procesora). Zapewnienie deterministycznego działania wątków współdzielących dane wymaga zastosowania mechanizmów synchronizacji danych.

**Mutex** – mechanizm synchronizacji, którego idea bazuje na blokowaniu dostępu do przestrzeni adresowej programu innym wątkom, pod warunkiem że sprawdzają one, czy mutex jest otwarty czy zamknięty. Poprawnie zrealizowana synchronizacja przy pomocy tego mechanizmu powinna zezwalać na dostęp do danych współdzielonych tylko jednemu wątkowi naraz.

**Pula obiektów** – wzorzec projektowy, którego założeniem jest dostarczenie pewnej ilości wstępnie zainicjowanych instancji obiektów, gotowych do pracy. W puli przechowywana jest pewna ilość już zainicjowanych obiektów, z których instancji może skorzystać inny obiekt. Celem podejścia jest redukcja czasu potrzebnego na stworzenie pewnego obiektu (np. uruchomienie wątku, alokację zasobów). Dzięki takiemu podejściu oprogramowanie jest w stanie zareagować w dużo krótszym czasie na zdarzenie. Pula obiektów, ze względu na swój charakter ma szczególne zastosowanie w sytuacjach, gdy czas potrzebny na stworzenie instancji obiektu jest relatywnie duży względem czasu potrzebnego na wykonanie na nim operacji.

**Całkowanie numeryczne metodą prostokątów** – numeryczna metoda obliczania całki oznaczonej na podstawie ciągu dyskretnych, pobieranych (w podanym zakresie) ze stałym krokiem próbek wartości wyjściowej funkcji. Definicja bazuje na definicji całki oznaczonej Riemanna, w której wartość całki interpretowana jest jako suma pól obszarów pod wykresem krzywej w zadanym przedziale. Metoda prostokątów bazuje na przybliżeniu tej sumy sumą pól prostokątów, których pole jest wynikiem iloczynu kroku całkowania i wartości funkcji w danym punkcie (1):

$$S_i = f(i) * dx \qquad (1)$$

gdzie

$S_i$ – pole powierzchni i-tego prostokąta, f(i) – wartość funkcji w i-tej próbce, dx – krok całkowania

Suma wszystkich pól w zadanym zakresie pozwala na numeryczne wyznaczenie całki oznaczonej funkcji, natomiast dokładność wyniku powinna rosnąć wraz ze zmniejszaniem kroku całkowania (2):

$$S = \sum_{i=1}^{n} S_i = f(1) * dx + f(2) * dx + \cdots + f(n) * dx, \qquad (2)$$

Równanie (2) można wyrazić także w postaci (3):

$$S = dx \sum_{i=1}^{n} f(i) \qquad (3)$$

## 3. Instrukcja laboratoryjna

W trakcie ćwiczenia należy zaimplementować pulę wątków, realizujących operację całkowania numerycznego funkcji jednej zmiennej metodą prostokątów. Wartości funkcji powinny zostać umieszczone w wektorze wartości typu *double*. Zakłada się, że wartości są próbkowane ze stałym krokiem, który jest także krokiem całkowania, podanym podczas wyzwolenia obliczeń. Do wykonania ćwiczenia niezbędne jest zastosowanie kodu przedstawionego na laboratorium 7.

1. Należy stworzyć klasę kalkulatora całki oznaczonej *Integrator*, dziedziczącą po interfejsie *IThread,* która powinna definiować metodę *ThreadRoutine* i działać w następujący sposób:
    a. W klasie muszą zostać zadeklarowane następujące elementy:
        i. publiczny typ wyliczeniowy Status (IDLE, WORKING)
        ii. prywatna kolekcja danych (*std::vector<double> data*), na podstawie których wyznaczana jest całka
        iii. prywatne pole *double step*, w którym przechowywana jest wielkość kroku całkowania
        iv. prywatne pole *std::mutex dataMutex*, do synchronizacji pól *data* oraz *step*
        v. prywatne pole *std::mutex statusMutex*, do synchronizacji pola *status*
        vi. prywatne pole *std::mutex resultMutex*, do synchronizacji pola *result*
        vii. prywatna, synchronizowana metoda dostępowa *void SetStatus(Status)*
        viii. publiczny konstruktor, inicjalizujący pola obiektu w bezpieczny sposób
        ix. publiczną, **synchronizowaną** metodę *void Count(const std::vector<double>&, const double)*, przyjmującą jako parametry referencję na dane wejściowe (wektor próbek dyskretnych) oraz krok całkowania

x. publiczne, **synchronizowane** metody dostępowe do stanu i wyniku, przechowywanych w obiekcie klasy *Integrator* (*Status GetStatus()*, *double GetResult()*)
xi. prywatną metodę *void ThreadRoutine()*, która stanowi definicję czysto wirtualnej metody z interfejsu *IThread*.

b. Metoda *ThreadRoutine*, powinna cyklicznie sprawdzać, czy stan obiektu nie zmienił wartości na *WORKING*. Można dodać w niej opóźnienie, tak aby sprawdzenie odbywało się co pewien czas (np. 1μs), jednak lepszym podejściem jest pozwolenie na realizację kolejnych wątków poprzez wywołanie funkcji *std::this_thread::yield()*. W przypadku, gdy prywatne pole *status* przyjmie wartość *WORKING,* metoda powinna zrealizować obliczanie całki dyskretnej metodą prostokątów. Po obliczeniu wyniku całkowania wątek powinien zapisać go w polu *result* i zmienić wartość *status* na *IDLE*.
c. Należy pamiętać, że dane o stanie obiektu, kolekcji danych, kroku całkowania i wyniku muszą być zawsze synchronizowane, czy to w metodach dostępowych, czy w metodzie opisującej działanie wątku.
d. Należy pamiętać, że całość synchronizacji danych powinna odbywać się wewnątrz obiektu klasy *Integrator*, wysoce niepożądana jest sytuacja, w której inny obiekt musiałby dbać o to, czy przypadkiem mutex nie jest zamknięty.

2. Należy stworzyć klasę puli wątków *IntegratorPool*, która powinna działać w następujący sposób:
    a. W klasie muszą zostać zadeklarowane następujące elementy:
        i. prywatna kolekcja wskaźników na dynamicznie alokowane obiekty klasy *Integrator* (*std::vector<Integrator *> pool*). Kolekcja stanowi pulę obiektów.
        ii. publiczny konstruktor, przyjmujący jako parametr ilość obiektów w puli.
        iii. publiczny destruktor, zatrzymujący wątki i zwalniający pamięć alokowaną w sposób dynamiczny.
        iv. publiczna metoda *Integrator * GetInstance()*, pozwalającą na zwrócenie pierwszej znalezionej instancji obiektu klasy *Integrator*, której stan jest równy wartości *Integrator::Status::IDLE*. Jeżeli wszystkie instancje są w trakcie wykonywania zadania, metoda powinna aktywnie oczekiwać do momentu, gdy zostanie wykryta pierwsza wolna instancja.
        v. Publiczna metoda *size_t GetLoad()*, pozwalająca na weryfikację liczby obiektów w puli, które są w trakcie realizacji zadania.
    b. Konstruktor klasy musi tworzyć pulę obiektów klasy *Integrator*, przechowywaną w prywatnej kolekcji *pool*.
    c. Wątki, realizowane w obiektach klasy *Integrator* muszą być uruchamiane w konstruktorze klasy *IntegratorPool*.
3. Należy przetestować działanie aplikacji w następujący sposób
    a. Wygenerować zbiór danych dyskretnych o długości 100 próbek typu *double*
    b. Utworzyć 100 obiektów klasy *Integrator*, uruchomić je i zlecić im zadanie całkowania zbioru testowego.

c. W następnej pętli poczekać na zakończenie obliczeń każdego z obiektów, zatrzymać wątki i usunąć obiekty, jeżeli były tworzone w sposób dynamiczny.
d. Zmierzyć sumaryczny czas realizacji podpunktów b i c. Jest to symulacja sytuacji, w której obsługa zadań w aplikacji jest realizowana w sposób konwencjonalny: utworzenie potrzebnego obiektu → przetwarzanie danych → usunięcie obiektu.
e. Utworzyć obiekt puli obiektów *IntegratorPool* zawierający 100 instancji obiektu klasy *Integrator*
f. Następnie 100-krotnie wydobyć z niego instancję obiektu klasy *Integrator*, której należy zlecić obliczenia całki numerycznej zbioru testowego z podpunktu a.
g. Poczekać na zakończenie pracy wszystkich obiektów w puli (można wykorzystać do tego metodę *GetLoad*).
h. Zmierzyć czas realizacji podpunktów f i g. Porównać wynik z czasem zmierzonym w punkcie d. Jest to symulacja sytuacji, gdy nie trzeba tworzyć za każdym razem instancji nowego obiektu do przetworzenia danych, tylko wykorzystuje się jeden z puli wstępnie zainicjowanych obiektów.
i. Powtórzyć zadania z punktów b-h dla różnych długości zbiorów danych (np. 1000, 100000) i różnych ilości obiektów w puli (np. 1, 10).
j. Otrzymane wyniki skomentować w pliku ***main.cpp***, który należy wysłać w sprawozdaniu, wraz z plikami klas ***IThread, Producer, Consumer, Integrator*** **oraz** ***IntegratorPool***.

Wskazówki:

1. Konieczne jest zapoznanie się bibliotekami standardowymi *mutex* i *thread* (http://en.cppreference.com/).
2. Do porównywania czasu działania fragmentów kodu można zastosować bibliotekę *std::chrono*. Przykład realizacji:

```
auto tstart = std::chrono::high_resolution_clock::now();
//tu umieść mierzony kod
auto tend = std::chrono::high_resolution_clock::now();
auto dur = std::chrono::duration_cast<std::chrono::microseconds>(tend-tstart).count();
```