PROGRAMOWANIE OBIEKTOWE – LABORATORIUM

ĆWICZENIE 7 - WIELOWĄTKOWOŚĆ I SYNCHRONIZACJA

---

1. Cel ćwiczenia

Celem ćwiczenia jest nabycie 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). Najważniejszymi zaletami wielowątkowości jest:

- zwiększenie stabilności działania aplikacji, poprzez możliwość wydzielenia krytycznych z punktu widzenia specyfikacji elementów procesu (np. obsługa zdarzeń klawiatury i myszy, zabezpieczenia aplikacji), które, w przypadku wąskiego gardła, np. konieczności przetworzenia dużej ilości danych, w aplikacji jednowątkowej mogłyby zostać zablokowane na czas przetwarzania tych danych;
- przyspieszenie działania aplikacji, np. poprzez podzielenie zasobów do przetworzenia na kilka wątków, działających współbieżnie lub wydzielenie elementów procesu, które mogą być realizowane niezależnie od siebie.

Każdy wątek w ramach jednego procesu współdzieli zasoby przestrzeń adresową przydzieloną całemu programowi. Pozwala to przede wszystkim na możliwość szybkiej, bezpośredniej wymiany danych między wątkami (w odróżnieniu od kilku procesów), w ramach pojedynczego procesu. Negatywnym efektem zastosowania wielowątkowości może być **utrata synchronizacji danych**, czyli sytuacja, w której dwa wątki działają współbieżnie na danych, będących niepełnym efektem ich działania, co może spowodować nieoczekiwane, błędne działanie aplikacji. Rozwiązaniem problemu są mechanizmy synchronizacji danych, takie jak np. mutex czy semafor.

Mutex – mechanizm synchronizacji, którego idea bazuje na blokowaniu dostępu do przestrzeni adresowej 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.

3. Instrukcja laboratoryjna

W trakcie ćwiczenia należy nabyć podstawowe umiejętności tworzenia wątków w klasie. Należy również zaimplementować algorytm synchronizacji oraz wymiany danych pomiędzy dwoma wątkami. W trakcie ćwiczenia należy wykonać następujące zadania:

1. Należy stworzyć interfejs do obsługi wątków, który stanowi klasa abstrakcyjna *IThread,* z wykorzystaniem biblioteki standardowej *thread*. Interfejs powinien implementować następujące elementy:
    a. Konstruktor i destruktor (wirtualny).
    b. Prywatny stan wątku - typ wyliczeniowy *Status* (STOPPED, RUNNING)
    c. Prywatny wskaźnik na obiekt wątku.
    d. Metodę publiczną, pozwalającą na wydobycie informacji o stanie wątku (bool IsRunning()), zwracającą *true*, gdy wątek jest uruchomiony.
    e. Metodę publiczną, pozwalającą na uruchomienie wątku (void Start()). Metoda powinna sprawdzać, czy wątek nie został już wcześniej uruchomiony.
    f. Metodę publiczną, pozwalającą na zatrzymanie wątku w sposób nagły (void Stop()).
    g. Metodę publiczną, pozwalającą na zatrzymanie wątku w sposób bezpieczny (void Join()).
    h. Metodę czysto wirtualną, opisującą działanie wątku (void ThreadRoutine()).
2. Należy stworzyć klasę producenta danych *Producer*, dziedziczącą po interfejsie *IThread,* która powinna definiować metodę *ThreadRoutine* i działać w następujący sposób:
    a. Wątek, opisany w metodzie *ThreadRoutine*, powinien co określony czas generować w sposób losowy jedną liczbę typu double.
    b. Dane wygenerowane przez wątek (metoda *ThreadRoutine*) powinny być dodawane na koniec prywatnej kolekcji, przechowywanej w obiekcie klasy *Producer*, synchronizowanej z wykorzystaniem biblioteki standardowej *mutex*.
    c. Klasa producenta musi posiadać synchronizowaną metodę dostępową, zwracającą aktualnie wygenerowaną kolekcję liczb losowych.
    d. Należy pamiętać, że całość synchronizacji danych powinna odbywać się wewnątrz obiektu producenta, wysoce niepożądana jest sytuacja, w której inny obiekt musiałby dbać o to, czy przypadkiem mutex nie jest zamknięty.
3. Należy stworzyć klasę konsumenta danych *Consumer*, dziedziczącą po interfejsie *IThread,* która powinna definiować metodę *ThreadRoutine* i działać w następujący sposób:
    a. Konstruktor klasy konsumenta musi przyjmować wskaźnik na obiekt producenta.
    b. Wątek, opisany w metodzie *ThreadRoutine*, powinien co określony czas pobierać kolekcję danych, wygenerowaną przez wskazanego producenta. Następnie, wątek ma wyliczyć średnią wszystkich wartości w kolekcji i zapisać ją w synchronizowanym (*mutex*), prywatnym polu klasy *Consumer*.
    c. Klasa konsumenta musi posiadać synchronizowaną metodę dostępową, zwracającą aktualną wartość średnią.

    d. Należy pamiętać, że całość synchronizacji danych powinna odbywać się wewnątrz obiektu konsumenta, wysoce niepożądana jest sytuacja, w której inny obiekt musiałby dbać o to, czy przypadkiem mutex nie jest zamknięty.
4. Przetestować zaimplementowany mechanizm w funkcji *main* następujący sposób:
    a. Uruchomić kilka wątków producentów i konsumentów.
    b. Zweryfikować ilość wątków, uruchomionych przez aplikację (w menedżerze zadań systemu).
    c. Zaimplementować mechanizm, pozwalający na oddziaływanie z wątkami:
        i. Wyświetlenie wartości średniej każdego z konsumentów (np. po wykryciu wciśnięcia klawisza *Enter*)
        ii. Zatrzymanie działania wątków (np. po wykryciu wciśnięcia klawisza *Esc*)

Wskazówki:

1. Konieczne jest zapoznanie się bibliotekami standardowymi *mutex* i *thread* (http://en.cppreference.com/).
2. Do generowania opóźnień w pracy wątków warto wykorzystać bibliotekę standardową *chrono*. Przykład realizacji uśpienia wątku na 100 [*ms*]:

```
std::this_thread::sleep_for(std::chrono::milliseconds(100));
```