Завдання 2.

Змоделювати паралельну роботу двох потоків (threads) зі спільною коміркою пам'яті (shared variable):

а) з використанням критичного сегменту (або atomic, mutex, lock, і т.п.)

б) без використання критичного сегменту.

Завдання а) використовує наступну функцію потоку, в якій використано захоплення м'ютексу:

```
void addMutex()
{
    for(int i = 0; i < Count; ++i)
    {
        lock_guard<mutex> guard(m_var);
        var = var + 1;
    }
}
```

Завдання б) виконує ті самі дії, але без м'ютексу. Крім того, цей код компілюється з вимкненою оптимізацією, аби завантаження, інкремент та збереження змінної були окремими командами процесора.

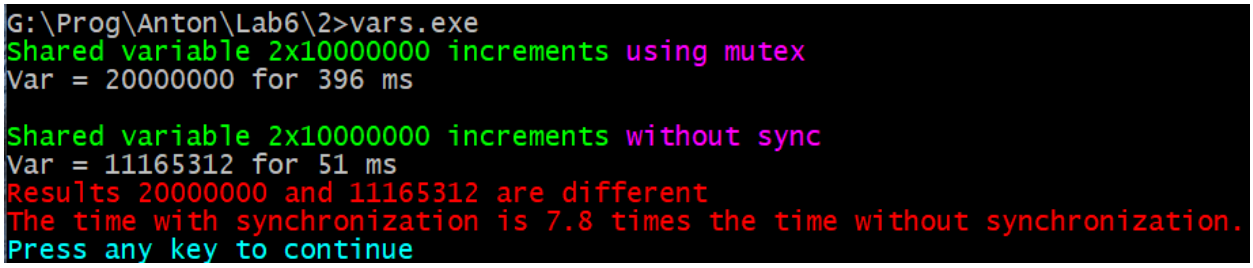
```
#pragma optimize("",off)
void addFree()
{
    for(int i = 0; i < Count; ++i)
    {
        var = var + 1;
    }
}
#pragma optimize("",on)
```

Асемблерний код:

```
    mov    eax, DWORD PTR ?var@@3HA        ; var
    inc    eax
    mov    DWORD PTR ?var@@3HA, eax        ; var
```

Завдання 2.1. *(2 бали) Продемонструвати різницю результатів обчислень у цих двох випадках. (Наприклад, збільшувати значення спільної комірки на 1: v=v+1; 10^9 разів в кожному потоці і дивитись результуюче значення v.)*

Як видно з наведеної копії екрану, при захисті операції з коміркою пам’яті м’ютексом отримано коректне значення 20000000. При відсутності синхронізації код виконується в 8 разів швидше, але дає помилкове значення. Це пов’язано в тим, що переключення між потоками можливе при виконанні трьох наведених вище асемблерних команд, тобто деякі операції інкременту можуть губитися (інший потік записую після повернення до виконання старе, менше значення).



Завдання 2.2. 2.2. *(+1 бал) Проаналізувати часову різницю різних варіантів реалізації та пояснити, чому іноді можливе отримання некоректного кінцевого результату (race condition).*

Час виконання коду без синхронізації приблизно на порядок менший, ніж з нею. Це пов’язано з тим, що певний чималий порівняно з часом на виконання інкременту час необхідний для виконання коду м’ютекса — його захоплення і звільнення.

Щодо некоректного результату, то розглянемо умову гонки. При тому, що два потоки звертаються до однієї змінної за допомогою наступного коду:

```
❶    mov    eax, DWORD PTR ?var@@3HA        ; var
❷    inc    eax
❸    mov    DWORD PTR ?var@@3HA, eax        ; var
```

можлива ситуація, коли потік 1 зчитав командою ❶ значення `var` — наприклад, 1000. В цей момент відбулося перемикання потоків, і тепер потік 2 виконує ці команди, записуючи в змінну `var` значення 1001. А якщо переключення відбувається не одразу після цього, а, наприклад, після 10 ітерацій, буде записано значення 1010. Переключення на потік 1 призведе до продовження виконання, тобто збільшення 1000 на 1 і запис у змінну

значення 1001. Усі збільшення, що за цей час відбулися в потоці 2, стають загубленими. Така поведінка призводить до некоректного результату.

Завдання 2.3. (+1 бал) *Спробувати досягти якомога швидшого результату при збереженні коректності обчислень (правильного фінального значення).*

Найшвидші результати можна отримати при використанні атомарних змінних, `atomic<int>`, операція інкременту для яких є атомарною, тобто не може бути перервана іншим потоком. Ще однією такою операцією є функція `fetch_add`. Ці методи демонструють більш швидке виконання, ніж у випадку використання м'ютекса, але повільніше, ніж без синхронізації:

```
Shared variable 2x10000000 increments using mutex
Var = 20000000 for 389 ms

Shared variable 2x10000000 increments without sync
Var = 15353017 for 48 ms
Results 20000000 and 15353017 are different
The time with synchronization is 8.1 times the time without synchronization.

Shared variable 2x10000000 fast increments using atomic ++
Var = 20000000 for 237 ms

Shared variable 2x10000000 fast increments using atomic feth_add
Var = 20000000 for 249 ms
Press any key to continue
```

Завдання 2.3.* *(+3 балів) Досягти варіанту, коли таке паралельне додавання виконується повністю синхронно, тобто, наприклад, 1000 додавань виконуються паралельно двома потоками крок-в-крок і збільшують значення спільної змінної від 0 до 1000. Тобто, не тільки кожний з двох паралельних потоків збільшує значення від 0 до 1000, а й обидва, запущені в паралель, також збільшують від 0 до 1000 (а не до 2000, як очікувалось би).*

Для цього нам треба вдатися до дуже дрібнозернистої оптимізації — на кожній ітерації, для того, щоб під час паралельної роботи втрачалося значення одного інкремента.

Для цього можна використовувати condition variable. Ось код, який виконує таку синхронізацію.

```
constexpr size_t Small = 1000;      // Number of summations for 2.3
size_t who = 0;                     // Thread id for switch
```

```
condition_variable chg;            // Condition variable to switch

// Variable increment function via intermediate value
// using condition_variable for thread switching
void addSync(unsigned int color)
{
    int save;               // Intermediate variable

    bool alone = false;     // Is single thread?

    for(int i = 0; i < Small; ++i)    // Small iteration
    {
        cout << attr(color) << '#';   // Color output what is thread active
        if (alone)                    // In single-thread mode don't use
        {                             // syncronization
            save = var;
            var = save + 1;
        }
        else
        {
            {
                unique_lock lck(m_var);
                save = var;           // Get original value of var
                who = _threadid;      // Write out current thread id
                chg.notify_one();     // Notify the thread about switch
            }
            {
                unique_lock lck(m_var);
                // Wait ANOTHER thread! if timeout,
                // the thread is considered to be the only thread
                // (single mode)
                if (chg.wait_for(lck,chrono::milliseconds(200),
                                 [&](){ return who != _threadid; }) == false)
                {
                    alone = true;
                }
                // Store new value of var
                var = save+1;
            }
        }
    }
}
```

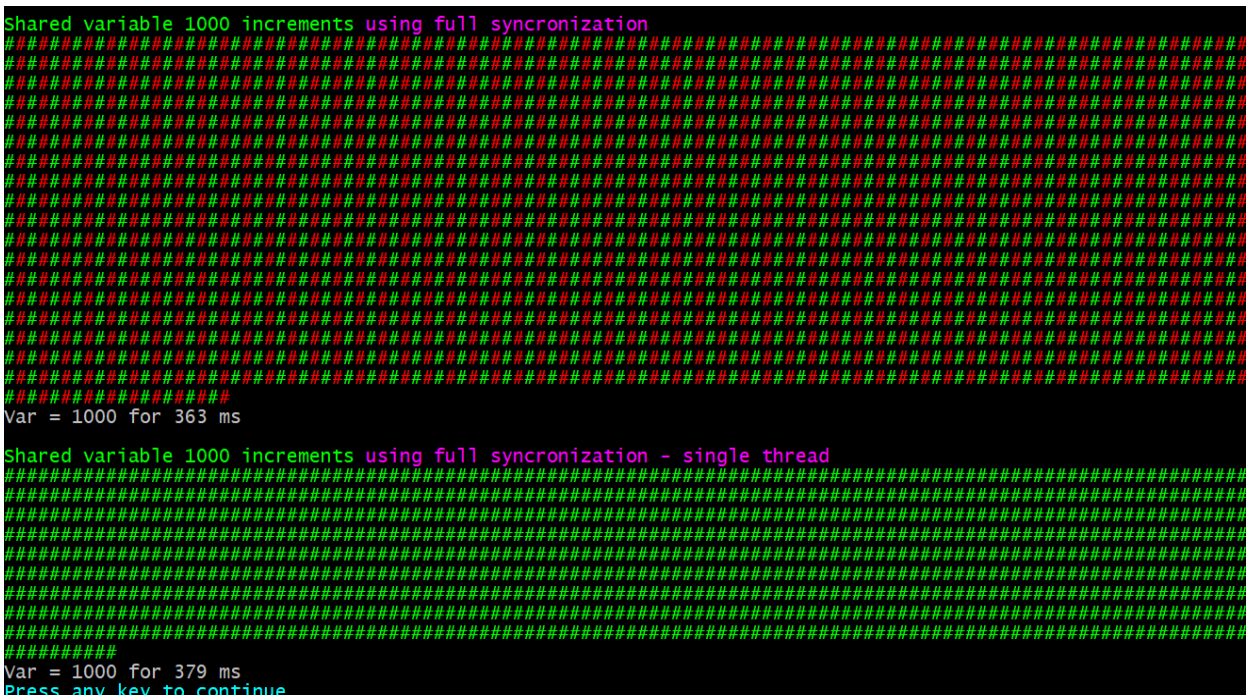
Якщо виконується робота в однопоточному режимі, значення змінної `alone` дорівнює `true`, і жодна синхронізація не використовується. Функція 1000 разів виконує збільшення спільної змінної:

```
            save = var;
            var = save + 1;
```

Якщо ж працюють два потоки, то потік завантажує значенні змінної в локальну змінну, записує свій ідентифікатор потоку в змінну `who`, та сповіщає інший потік через змінну `chg` про можливість роботи. Цей інший потік активується, якщо ідентифікатор потоку в змінній `who` відрізняється від його ідентифікатору (що запобігає можливості продовження роботи

першим потоком). Другий потік записує збільшену змінну, і повторює дії першого потоку. Виконується переключення на перший потік, який теж записує таке саме значення.

Якщо відбувається таймаут чекання активації іншого потоку, це означає, що другого потоку немає, і працює лише один потік. Тоді змінна `alone` отримує значення `true`, щоб потоку більше не доводилося займатися синхронізацією. Результат роботи в двопоточному та однопоточному режимі показано нижче. Кольоровий вивід на екран дає змогу побачити степінь синхронізації наочно.



Таким чином, завдання повністю виконане. В обох режимах потік виконує 1000 збільшень змінної.