

СИНХРОНИЗАЦИЯ

- Под *синхронизацией потоков* понимается исполнение этими потоками условных непрерывных действий. Рассмотрим частные случаи синхронизации, которые наиболее часто встречаются на практике.
- Если оператор **await** имеет следующий вид:

await(логическое условие);

- то он просто ждет оповещения о выполнении некоторого логического условия. Этот случай называется *условной синхронизацией*, а само логическое условие также называется *событием*. В этом случае часто говорят, что оператор **await** ждет наступления некоторого события.
- Если оператор **await** имеет вид:

await(TRUE) действие;

- то происходит безусловное выполнение непрерывного действия. Этот случай называется *взаимным исключением*, а программный код, исполняемый внутри непрерывного действия, называется *критической секцией*.

Задача условной синхронизации

- Для постановки задачи рассмотрим два потока `thread_1` и `thread_2`, которые работают следующим образом. Поток `thread_1` выполняет некоторые действия, а затем ждет наступления события `event`, после которого выполняет другие действия. В свою очередь поток `thread_2` также выполняет некоторые действия, а после их завершения оповещает поток `thread_1` о наступлении события `event`. Затем поток `thread_2` выполняет оставшиеся действия. Такая синхронизация работы потоков и называется задачей условной синхронизации.

```
bool event = false;    // событие event
void thread_1()        // поток thread_1
{
actions_before_event(); // действия до наступления события
while(!event);         // ждем, пока событие не произошло
actions_after_event(); // действия после наступления события
}
void thread_2()        // поток thread_2
{
some_actions();        // действия, о которых оповещает событие
event = true;          // отмечаем о наступлении события
other_actions();       // действия, происходящие после события
}
```

Задача взаимного исключения

- Чтобы упростить рассуждения, эта задача будет сформулирована только для двух параллельных потоков. Сначала предположим, что два параллельных потока работают с одним и тем же ресурсом, который в этом случае называется *разделяемым* или *совместно используемым ресурсом*. Далее считаем, что в каждом потоке программный код, который осуществляет доступ к этому ресурсу, заключен в свою критическую секцию. Тогда задача взаимного исключения для двух потоков может быть сформулирована следующим образом: обеспечить двум потокам взаимоисключающий доступ к некоторому совместно используемому ресурсу.

Решение этой задачи должно удовлетворять следующим требованиям:

- ✓ **требование безопасности** — в любой момент времени в своей критической секции может находиться только один поток;
- ✓ **требование поступательности** — потоки не могут блокировать работу друг друга, ожидая разрешения на вход в критическую секцию;
- ✓ **требование справедливости** — каждый поток получает доступ в критическую секцию за ограниченное время.

```
bool x1 = false; bool x2 = false;
int q;      // номер потока, которому предоставляется очередь входа в критическую секцию
void thread_1() // поток thread_1
{
while(true)
{
non_critical_section_1(); // код вне критической секции
x1 = true;      // поток thread_1 хочет войти в критическую секцию
q = 2;         // предоставить очередь потоку thread_2
while(x2 && q == 2); // ждем, пока в критической секции находится поток thread_2
critical_section_1(); // входим в критическую секцию
x1 = false;    // поток thread_1 находится вне критической секции
}
}
```

```
void thread_2()          // поток thread_2
{
while(true)
{
non_critical_section_2(); // код вне критической секции
x2 = true;               // поток thread_2 хочет войти в критическую секцию
q = 1;                   // предоставить очередь потоку thread_1
while(x1 && q == 1);      // ждем, пока в критической секции находится поток thread_1
critical_section_2();    // входим в критическую секцию
x2 = false;              // поток thread 2 находится вне критической секции
}
}
```


Примитивы синхронизации

- Примитивом синхронизации называется программное средство высокого уровня для решения задач синхронизации. Обычно примитивы синхронизации реализованы как объекты ядра операционной системы, которые предназначены для решения задач синхронизации потоков и процессов.

Примитив синхронизации *condition* (условие)

- Ниже приведен класс **condition**, определяющий одноименный примитив:

```
class Condition {  
    bool event ;  
    ThreadQueue tq; // очередь потоков  
public:  
    Condition(bool b): event(b) {} // конструктор  
    ~Condition() {} // деструктор  
    void Signal() // сигнализировать о том, что условие  
        выполнено {
```

```
disable_interrupt();           // запрещаем прерывания
if(!tq.DequeueThread())
event = true; enable_interrupt(); // разрешаем прерывания
void Wait(Thread t)           // ждать выполнения условия
{
disable_interrupt();           // запрещаем прерывания
if(event)
event = false;                 // сбрасываем условие
else
tq.EnqueueThread(t);           // ставим поток в очередь ожидания
enable_interrupt();           // разрешаем прерывания
}
};
```

Примитив синхронизации *semaphore* (семафор)

Ниже приведен класс `Semaphore`, определяющий одноименный примитив.

```
class Semaphore
{
    unsigned counter;           // счетчик
    ThreadQueue tq;            // очередь потоков
public:
    Semaphore(unsigned n): counter(n) {} // конструктор
    ~Semaphore() {}              // деструктор

    void Signal() // сигнализировать о том, что условие выполнено
    {
        disable_interrupt(); // запрещаем прерывания
        if(!tq.DequeueThread())
            ++counter;
        enable_interrupt(); // разрешаем прерывания
    }

    void Wait(Thread t) // ждать выполнения условия
```

```
disable_interrupt(); // запрещаем прерывания
if(counter > 0)
    --counter; // сбрасываем условие
else
    tq.EnqueueThread(t); // ставим поток в очередь ожидания
enable_interrupt(); // разрешаем прерывания
}
};
```