Средства синхронизации на платформе .NET (C#)

Блокировка	**Join, Sleep, SpinWait**
Взаимно-исключительный доступ	**lock, Monitor, Mutex, SpinLock**
Сигнальные сообщения	**AutoResetEvent, ManualResetEvent, ManualResetEventSlim**
Семафоры	**Semaphore, SemaphoreSlim**
Атомарные операторы	**Interlocked**
Сценарии синхронизации	**Barrier, CountdownEvent, ReaderWriterLock, ReaderWriterLockSlim**

Типы ожидания

- Активное ожидание – циклическая проверка статуса ожидаемого события

  while(!thr.IsAlive) ;

- Пассивное ожидание – выгрузка потока

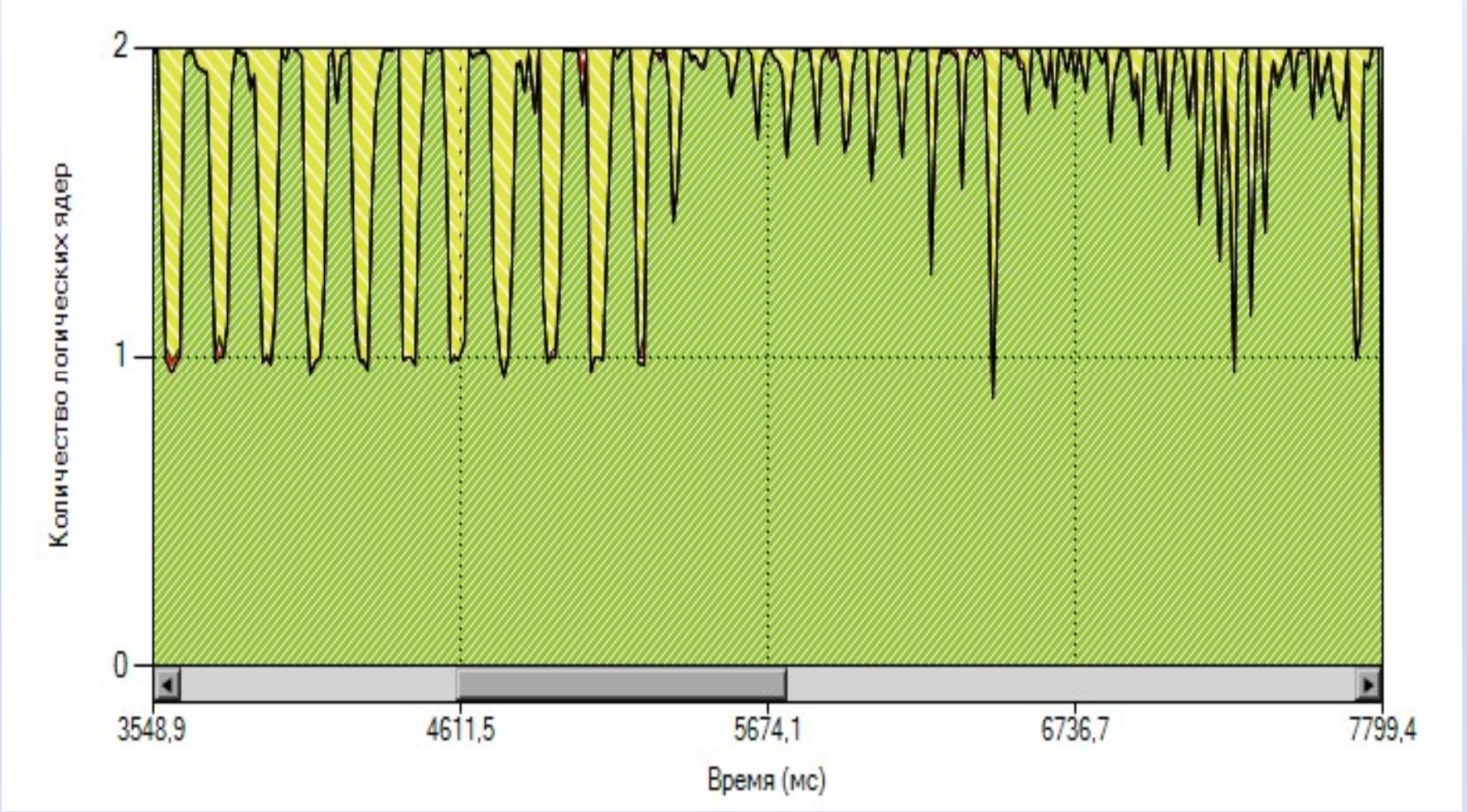
  thr.Join();

- Гибридное ожидание

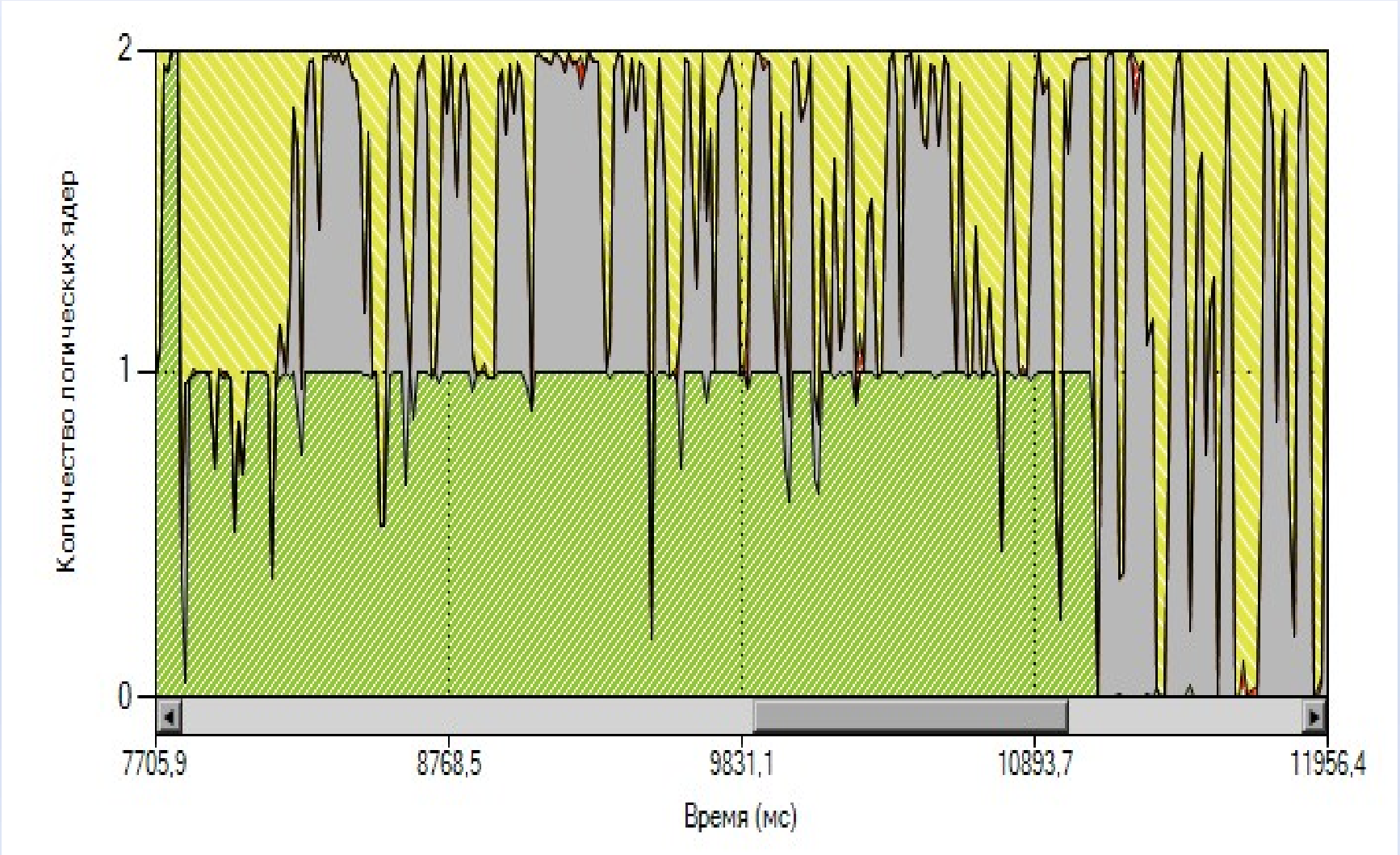
  SpinWait.SpinUntil(() => !thr.IsAlive);

  while(!b) Thread.Sleep(100);

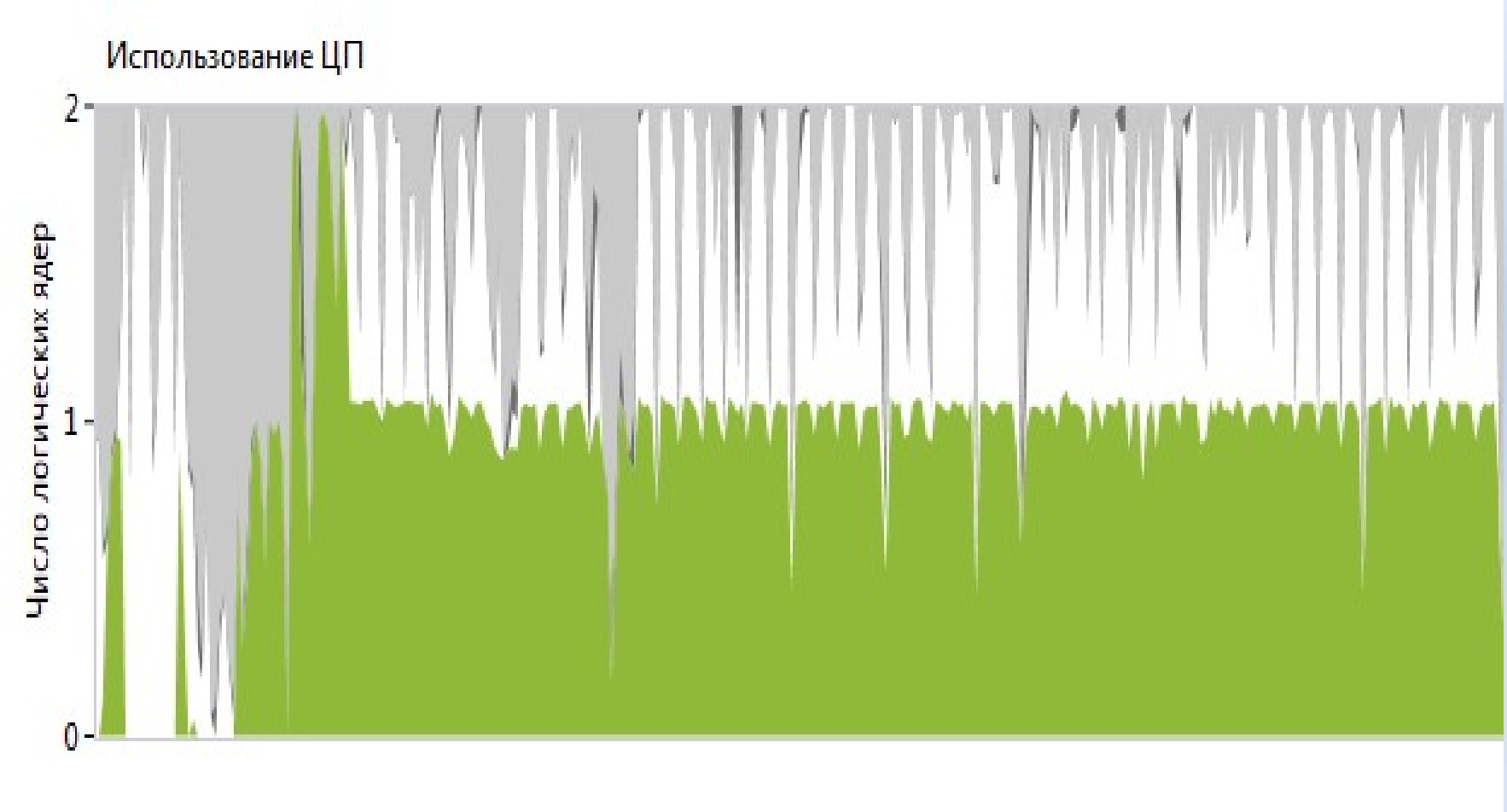
«Активное» ожидание



Пассивное ожидание



Гибридное ожидание



- Ожидание может быть активным или пассивным. При активном «ожидании» поток циклически проверяет статус ожидаемого события.

```
Thread thr = new Thread(SomeWork);
thr.Start();
while(thr.IsAlive) ;
```

Такая блокировка называется активной, так как фактически поток не прекращает своей работы и не освобождает процессорное время для других потоков. Активное ожидание эффективно ***только*** при незначительном времени ожидания.

- Пассивное ожидание реализуется с помощью операционной системы, которая сохраняет контекст потока и выгружает его, предоставляя возможность выполняться другим потокам. При наступлении ожидаемого события операционная система «будит» поток – загружает контекст потока и выделяет ему процессорное время. Пассивное ожидание требует времени на сохранение контекста потока при блокировке и загрузку контекста при возобновлении работы потока, но позволяет использовать вычислительные ресурсы во время ожидания для выполнения других задач.

Средства для взаимного исключения

```
lock(sync_obj)
{
    // Критическая секция
}
```

sync_obj – объект синхронизации, используется как идентификатор секции; может быть только ссылочным объектом

• В качестве идентификатора критической секции используется «пустой» объект типа Object.

• В каждом потоке, в котором есть критическая секция, должен использовать тот же самый объект sync_obj

```
object    sync_obj  = new object();
..
lock(sync_obj)
{
    // Критическая секция
}
```

• В качестве идентификатора критической секции используется объект, связанный с разделяемым ресурсом.

• Нет потребности в дополнительных объектах синхронизации

```
..
lock(data)
{
    // Критическая секция
    data.x++;
}
```

• В качестве идентификатора критической секции используется строковая константа.

```
..
lock(“This is critical section, so enter one-by-one”)
{
    // Критическая секция
}
```

- Не рекомендуется использовать ссылку this
- Нельзя использовать объекты значимых типов

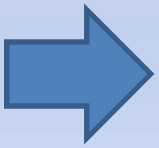
```
// Не рекомендуется!
lock(this)
{
    // Критическая секция
}
// Нельзя использовать  значимые типы (ValueTypes)
lock('d') { .. }
lock(3) { .. }
lock(myStructData) { .. }
```

Monitor

- Применяется для обеспечения взаимно-исключительного доступа к фрагменту кода
- Реализует «условные» входы в критическую секцию
- Обладает возможностями условной синхронизации с ожидающими потоками

Применение монитора

```
lock(sync_obj)
{
    // Critical section
}
```



```
try
{
    Monitor.Enter(sync_obj);
    // Critical section
}
finally
{
    Monitor.Exit(sync_obj);
}
```

«Попытка» входа в критическую секцию

```
b = Monitor.TryEnter(sync_obj);
if(!b)
{
    // Выполняем полезную работу
    DoWork();
    // Снова пробуем войти в критическую секцию
    Monitor.Enter(sync_obj);
}
// Критическая секция
ChangeData();
// Выходим
Monitor.Exit(sync_obj);
```

«Попытка» входа в критическую секцию

```
while(!Monitor.TryEnter(sync_obj, 100))
{
    // Полезная работа
    DoWork();
}
// Критическая секция
ChangeData();
// Выходим
Monitor.Exit(sync_obj);
```

Сигнальный механизм объекта Monitor

- **Monitor.Wait**(P) – приводит к блокировке текущего потока, выполняющего критическую секцию; один из ожидающих потоков может войти в секцию P.
- **Monitor.Pulse**(P) – сигнал для заблокированного потока о возможности продолжить выполнение критической секции после её освобождения текущим потоком

Проблема взаимоблокировки

- Потоки работают с несколькими ресурсами
- Доступ к каждому ресурсу должен быть монопольным
- Порядок захвата ресурсов приводит к тому, что каждый поток владеет одним ресурсом и требует другого для продолжения

```
void ThreadOne()
{
        Monitor.Enter(P);
        if(!Monitor.TryEnter(Q))
        {
                // Если Q занят другим потоком, освобождаем P и
                // ожидаем завершения работы потока
                Monitor.Wait(P);
                // Освободился ресурс P, смело захватываем и Q
                Monitor.Enter(Q);
        }
        // Теперь у потока есть и P, и Q, выполняем работу
        ..
        // Освобождаем ресурсы в обратной последовательности
        Monitor.Exit(Q);
        Monitor.Exit(P);
}
```

```
void ThtreadTwo()
    {
        Monitor.Enter(Q);
        Monitor.Enter(P);
        // Выполняем необходимую работу
        ..
        // Обязательный сигнал для потока, который
        // заблокировался при вызове Monitor.Wait(P)
        Monitor.Pulse(P);
        Monitor.Exit(P);
        Monitor.Exit(Q);
    }
```