Объекты ядра Windows

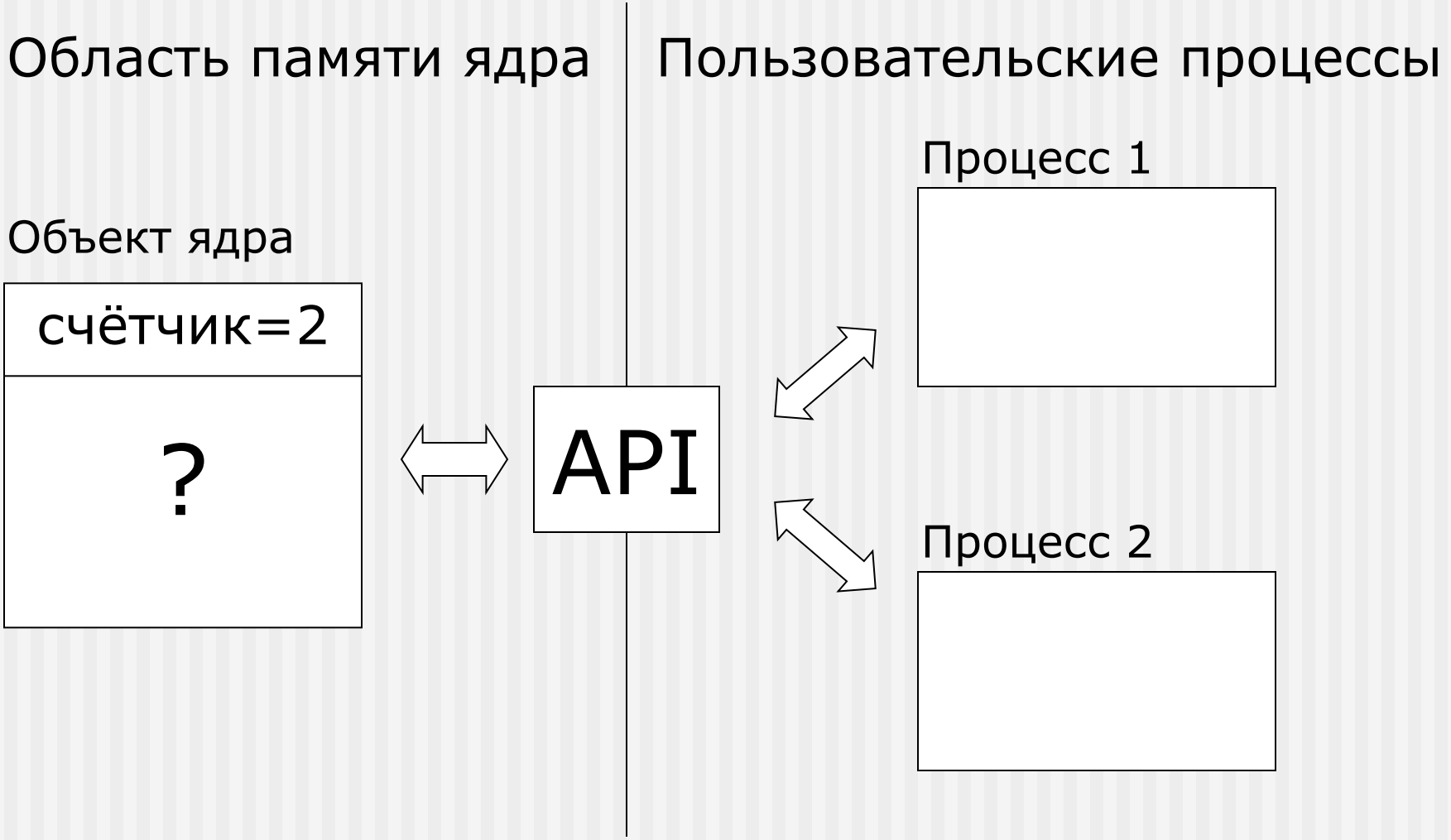
Типы объектов ядра

- маркеры доступа / access token
- события / event
- файлы / file
- проекции файлов / file mapping
- порты завершения ввода-вывода / I/O completion port
- задания / Job
- почтовые ящики / mailslot
- мъютексы / mutex
- каналы / pipe
- процессы / process
- семафоры / semaphore
- потоки / thread
- ожидаемые таймеры / waitable timer

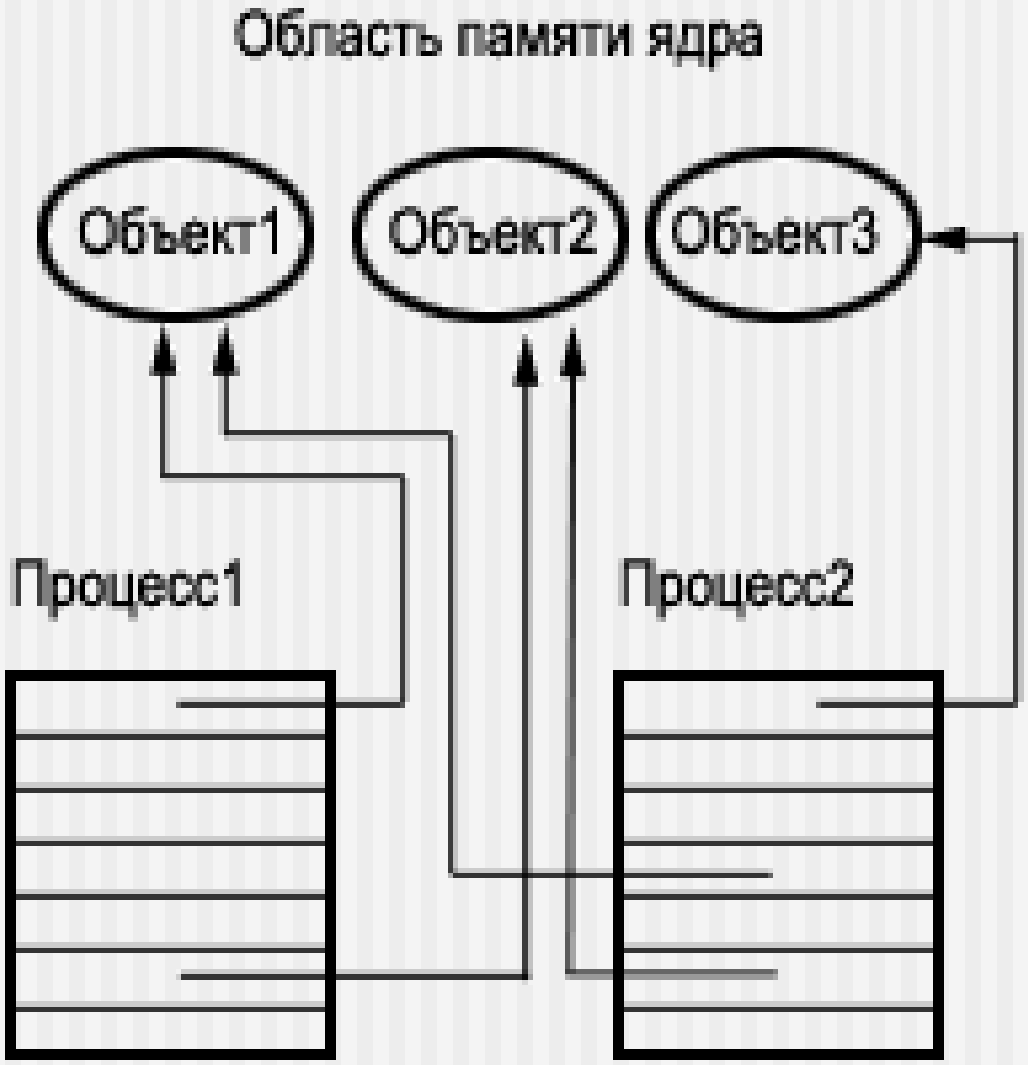
Защита объектов ядра

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD  nLength;
           // размер структуры
  LPVOID lpSecurityDescriptor;
           // дескриптор защиты
  BOOL   bInheritHandle;
           // наследовать объект
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES;
```

Пользователи объектов ядра



Таблицы дескрипторов



Создание объектов ядра

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
     // имя файла
  DWORD dwDesiredAccess,
     // права доступа к файлу
  DWORD dwShareMode,
     // режим совместного использования
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
     // права доступа к объекту ядра
  DWORD dwCreationDisposition,
     // режим создания
  DWORD dwFlagsAndAttributes,
     // аттрибуты файла
  HANDLE hTemplateFile
     // дескриптор шаблона
);
```

Создание объектов ядра

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
      // права доступа к объекту ядра
  LONG lInitialCount,
      // начальное значение
  LONG lMaximumCount,
      // максимальное значение
  LPCTSTR lpName
      // имя объекта
 );
```

Закрытие объектов ядра

```
BOOL CloseHandle(
   HANDLE hObject
      // дескриптор объекта
);
```

Совместное использование

- Наследование дескрипторов объектов ядра
- Копирование дескрипторов объектов ядра
- Имена объектов ядра

Наследование объектов ядра

Копирование дескрипторов

```
BOOL DuplicateHandle(
  HANDLE hSourceProcessHandle,
  // дескриптор процесса-источника
  HANDLE hSourceHandle,
     // дескриптор объекта который копируем
  HANDLE hTargetProcessHandle,
     // дескриптор процесса-получателя
  LPHANDLE lpTargetHandle,
    // дескриптор объекта-копии
  DWORD dwDesiredAccess,
    // режим доступа
  BOOL bInheritHandle,
    // режим наследования копии
  DWORD dwOptions
     // дополнительные опции
);
```

Имена объектов ядра

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
      // права доступа к объекту ядра
  LONG lInitialCount,
      // начальное значение
  LONG lMaximumCount,
      // максимальное значение
  LPCTSTR lpName
      // имя объекта
 );
```

Имена объектов ядра

```
HANDLE OpenSemaphore(
   DWORD dwDesiredAccess,
       // режим доступа к объекту ядра
   BOOL bInheritHandle,
       // режим наследования
   LPCTSTR lpName
       // имя объекта
);
```

Средства для синхронизации в Windows

- **Объекты синхронизации и Wait-функции**
- Interlocked функции
- Критические секции

Объекты синхронизации

Объекты синхронизации

- Событие / Event
- Семафор / Semaphore
- Мьютекс / Mutex
- Ожидаемый таймер / Waitable Timer

Состояния:

- Свободное / Signaled
- Занятое / Non-signaled

Другие объекты, с которыми работают Wait-функции

- Поток / Thread
- Процесс / Process

Wait-функции

```
DWORD WaitForSingleObject(
  HANDLE hHandle,          // дескриптор объекта
  DWORD dwMilliseconds     // время ожидания
);
```

Возвращаемое значение:

WAIT_OBJECT_0 – Объект свободен
WAIT_TIMEOUT – Время вышло, объект занят
WAIT_FAILED - Ошибка

Wait-функции

```
DWORD WaitForMultipleObjects(
  DWORD nCount,                  // число объектов
  CONST HANDLE *lpHandles,   // массив дескрипторов
  BOOL bWaitAll,                 // все или одно
  DWORD dwMilliseconds       // время ожидания
);
```

Возвращаемое значение:

WAIT_OBJECT_0 – Первый объект свободен
WAIT_OBJECT_0+1 – Второй объект свободен
…
WAIT_TIMEOUT – Время вышло, все объекты занят
WAIT_FAILED - Ошибка

Событие / Event

Содержание объекта:

Булева переменная: свободно/занято

Сделать свободным:

```
BOOL SetEvent(HANDLE hEvent);
```

Сделать занятым:

```
BOOL ResetEvent(HANDLE hEvent);
```

Создание события

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes, // SD
  BOOL bManualReset,     // тип события
  BOOL bInitialState,    // начальное состояние
  LPCTSTR lpName         // имя события
);
```

Событие с автосбросом (bManualReset==FALSE) автоматически переходит в занятое состояние если его «дождались»

- побочный эффект ожидания

Событие - пример

```
HANDLE hEvent;

hEvent=CreateEvent(NULL,FALSE,FALSE,NULL);
```

Поток 1

```
if(WaitForSingleObject
        (hEvent,INFINITE)
            ==WAIT_OBJECT_0)

{
  printf("Event signaled");
  ResetEvent(hEvent);
}
```

Поток 2

```
printf("press any key");
getchar();
SetEvent(hEvent);
```

Семафор / Semaphore

Содержание объекта:

Счётчик: 0 – объект занят

>0 – объект свободен

Увеличить счётчик:

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,        // дескриптор объекта
    LONG lReleaseCount,       // сколько добавить
    LPLONG lpPreviousCount    // предыдущее значение
);
```

Уменьшить счётчик:

Уменьшается на 1 как побочный эффект ожидания

Создание семафора

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // SD
  LONG lInitialCount,    // начальное значение
  LONG lMaximumCount,    // максимальное значение
  LPCTSTR lpName         // имя объекта
);
```

Значение семафора не может стать больше максимального.

Если значение ReleaseCount при вызове ReleaseSemaphore слишком велико состояние семафора не изменяется.

Мьютекс / Mutex

Содержание объекта:

Идентификатор потока-хозяина

Сделать занятым (захватить):

Побочный эффект ожидания

Сделать свободным:

```
BOOL ReleaseMutex(HANDLE hEvent);
```

Создание мьютекса

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,  // SD
  BOOL bInitialOwner,     // захватить при создании
  LPCTSTR lpName          // имя объекта
);
```

Использование мьютекса

```
int count=0;
int data[10];
HANDLE hMutex;

void main(void)
{
   hMutex=CreateMutex(NULL,FALSE,NULL);
                …
   CloseHandle(hMutex);
}

void StoreData(int element)
{
  WaitForSingleObject(hMutex,INIFINITE);
  data[count]=element;
  count++;
  ReleaseMutex(&hMutex)
}
```

Waitable Timer

Содержание объекта:

Таймер – активизируется в заданное время.

Создание:

```
HANDLE CreateWaitableTimer(
  LPSECURITY_ATTRIBUTES lpTimerAttributes, // SD
  BOOL bManualReset,       // ручной или автосброс
  LPCTSTR lpTimerName      // имя
);
```

Сброс:

Автосброс или

`BOOL CancelWaitableTimer(HANDLE hTimer);`

Установка таймера

```
BOOL SetWaitableTimer(
  HANDLE hTimer,                    // дескриптор таймера
  const LARGE_INTEGER *pDueTime, // время первого срабатывания
  LONG lPeriod,                     // интервал
  PTIMERAPCROUTINE pfnCompletionRoutine, // completion routine
  LPVOID  lpArgToCompletionRoutine,      //  completion  routine
   parameter
  BOOL fResume                      // включить питание ПК
);

pDueTime – время срабатывания
               Если положительное – абсолютное время
               Если отрицательное – через заданный период

lPeriod  - период срабатывания в миллисекундах
               0 – только одно срабатывание
```

Средства для синхронизации в Windows

- Объекты синхронизации и Wait-функции
- **Критические секции**
- Interlocked функции

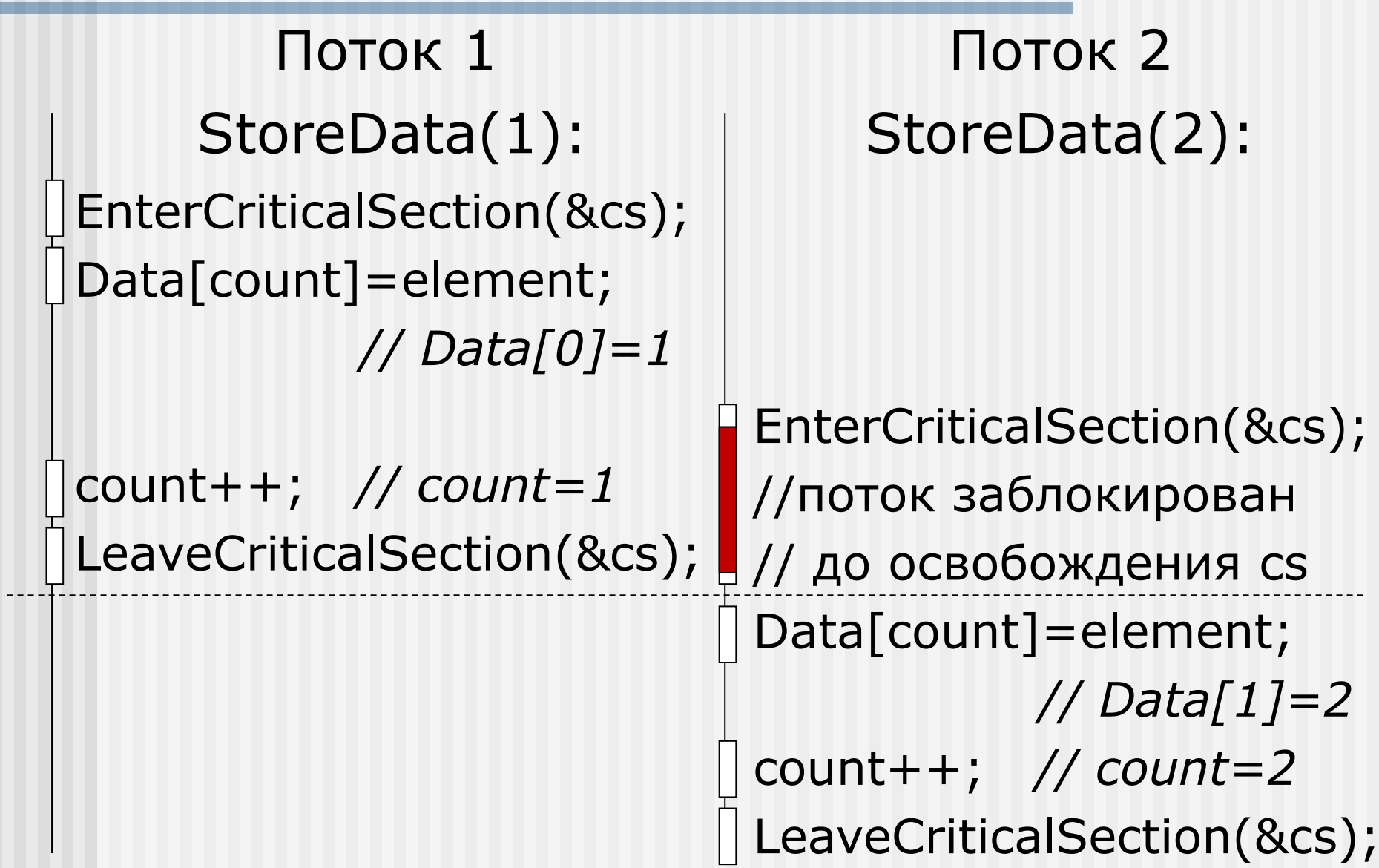
Критические секции

```
VOID InitializeCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
// инициализация критической секции

VOID EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
// захват критической секции

VOID LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
//освобождение критической секции

VOID DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
// удаление критической секции
```

Критическая секция - пример

```
int count=0;
int data[10];
CRITICAL_SECTION cs;

void main(void)
{
        InitializeCriticalSection(&cs);
        …
        DeleteCriticalSection(&cs);
}

void StoreData(int element)
{
        EnterCriticalSection(&cs);
        data[count]=element;
        count++;
        LeaveCriticalSection(&cs);
}
```

Критическая секция - пример



Если мы не хотим ждать

```
BOOL TryEnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);


CRITICAL_SECTION cs;

void SomeFunction(void)
{
      if(TryEnterCriticalSection(&cs))
      {
            … // работаем с ресурсом
            LeaveCriticalSection(&cs);
      }
      else
            // не работаем с ресурсом
}
```

Если мы не хотим ждать

```
BOOL TryEnterC
  LPCRITICAL_S


CRITICAL_SECTI

void SomeFunct
{
      if(TryEr
      {
            … // работаем с ресурсом
            LeaveCriticalSection(&cs);
      }
      else
            // не работаем с ресурсом
}
```

Псевдокод функции TryEnterCriticalSection:

```
bool TryEnterCriticalSection(CS)
{
        if(!Занята(CS))
        {
                EnterCriticalSection(CS);
                return true;
        }
        else
                return false;
}
```

Выполняется атомарно.

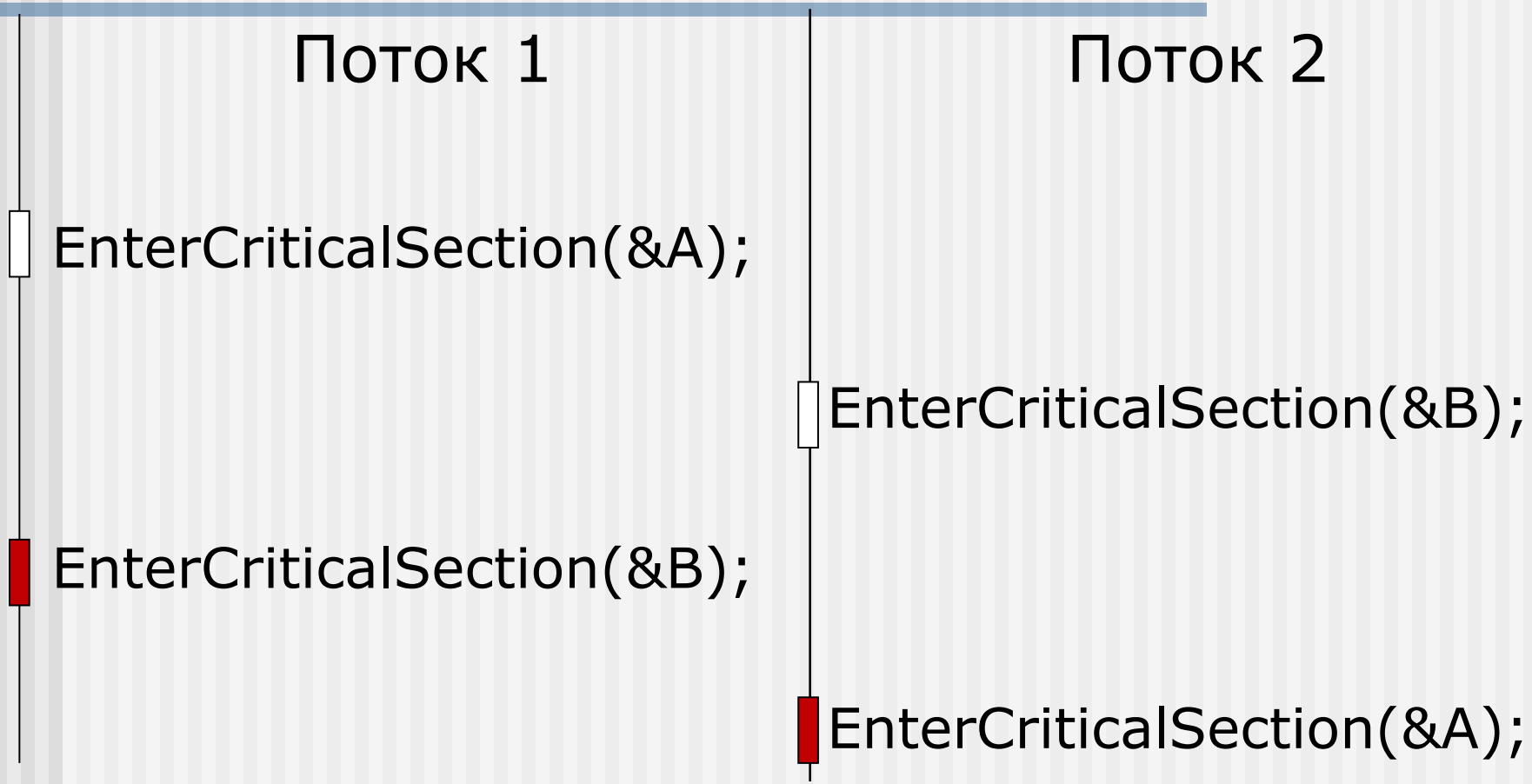
Если мы не хотим ждать

```
BOOL TryEnterC
  LPCRITICAL_S


CRITICAL_SECTI

void SomeFunction(void)
{
      if(TryEnterCriticalSection(&cs))
      {
            … // работаем с ресурсом
            LeaveCriticalSection(&cs);
      }
      else
            // не работаем с ресурсом
}
```

Неправильный вариант использования:

```
if(TryEnterCriticalSection(&cs))
{
      EnterCriticalSection(&cs);
      … // работаем с ресурсом
      LeaveCriticalSection(&cs);
}
```

Рано радоваться



Мьютексы и критические секции

```
int count=0;
int data[10];
CRITICAL_SECTION cs;

void main(void)
{
   InitializeCriticalSection(&cs);
            …
   DeleteCriticalSection(&cs);
}

void StoreData(int element)
{
   EnterCriticalSection(&cs);
   data[count]=element;
   count++;
   LeaveCriticalSection(&cs);
}
```

```
int count=0;
int data[10];
HANDLE hMutex;

void main(void)
{
  hMutex=CreateMutex(NULL,FALSE,NULL);
            …
  CloseHandle(hMutex);
}

void StoreData(int element)
{
  WaitForSingleObject(hMutex,INIFINITE);
  data[count]=element;
  count++;
  ReleaseMutex(&hMutex)
}
```

Мьютексы и критические секции

Мьютекс	Критическая секция
Объект ядра Может использоваться из разных процессов Можно использовать все возможности Wait-функций.	Объект программы Быстрее работает