Объекты ядра Windows

Объект ядра

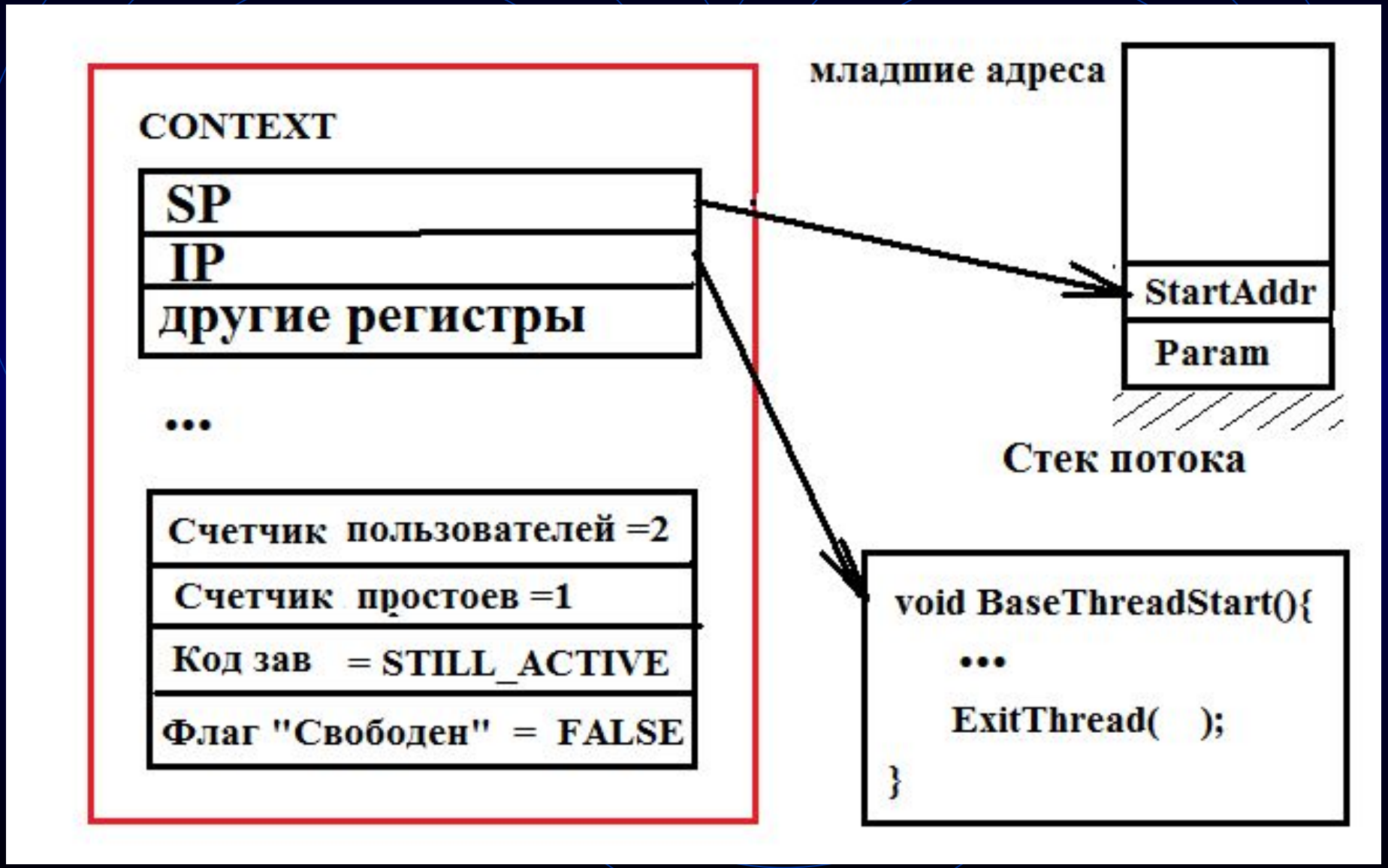
Объекты ядра принадлежат ядру, а не процессу. Процесс может использовать объект ядра с помощью описателей (HANDLE)

Объект ядра – это структура памяти, выделенный ядром и доступный только ему. В объекте ядра содержится информация :

- общая для всех (Дескриптор защиты, Счетчик числа пользователей, …)

- специфичная для данного типа объектов (объект «процесс» - идентификатор, базовый приоритет, код завершения, объект «семафор» - имя семафора, текущее состояние, …)

Объект ядра «Поток»



HANDLE Obj = ...

1. Obj=CreateThread(…);
2. Obj = CreateFileMapping(…);
3. Obj = CreateSemaphore(…);
4. Obj = CreateMutex(…);
5. CloseHandle(Obj);
6. Obj = OpenSemaphore(…);

…

```
static HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpsa,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE pfnThreadProc,
    void* pvParam,
    DWORD dwCreationFlags,
    DWORD* pdwThreadId ) ;
```

Таблица описателей, принадлежащих процессу

Индекс	Указатель на блок памяти объекта ядра	Маска доступа с набором битовых флагов (DWORD)	флаги
1	0x????????	0x????????	0x????????
2	0x00000000	Закрыт объект	
HANDLE obj = …	Счетчик пользователей …		Из LPSECURITY_ATTRIBUTES

Индекс в дочернем процессе (CreateProcess)			
1			

Таблица описателей, принадлежащих процессу

Закрытие описателя **(HANDLE)** процесса или потока не заставляет систему уничтожить этот процесс или поток.

Закрывая описатель, программа просто сообщает системе, что статистические данные для этого процесса или потока ей больше не нужны, но процесс или поток продолжает исполняться системой до тех пор, пока он сам не завершит себя.

Таблица описателей, принадлежащих процессу

Система способна повторно использовать идентификаторы процессов и потоков.

При создании процесса система формирует объект "процесс", присваивая объекту идентификатор с некоторым значением. Создавая новый объект "процесс", система уже не присвоит ему данный идентификатор.

Но после выгрузки из памяти первого объекта следующему создаваемому объекту "процесс" может быть присвоен тот же идентификатор .

Совместное использование несколькими процессами объекта ядра

1. Через общий HANDLE
2. Через наследование дочерними процессами объектов родительских процессов
2. Использование именованных объектов

```
HANDLE SemEnd =
    OpenSemaphore(SEMAPHORE_ALL_ACCESS,
true, "nameOfSemEnd");
if (SemEnd==NULL)    SemEnd =
    CreateSemaphore(NULL,0,1,"nameOfSemEnd");
  else MessageBox(…"Кто-то создал семафор!");
```

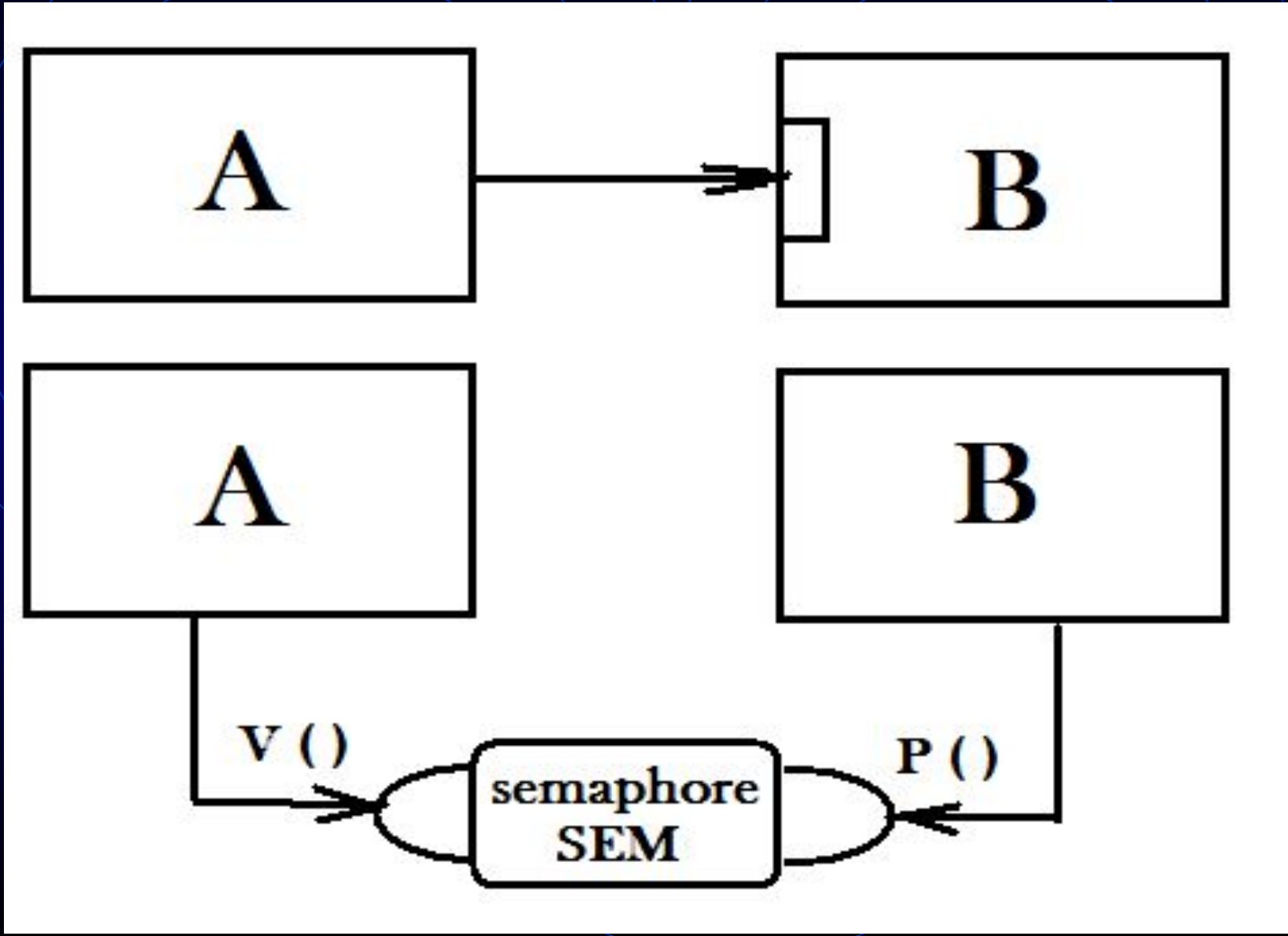
Именование объекта при его создании

```
HANDLE WINAPI    CreateSemaphore(
__in LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
__in LONG lInitialCount,
__in LONG lMaximumCount,
__in LPCTSTR lpName );

HANDLE WINAPI    CreateFileMapping(
 __in HANDLE hFile,
__in LPSECURITY_ATTRIBUTES lpAttributes,
 __in DWORD flProtect,
 __in DWORD dwMaximumSizeHigh,
 __in DWORD dwMaximumSizeLow,
 __in LPCTSTR lpName );
```

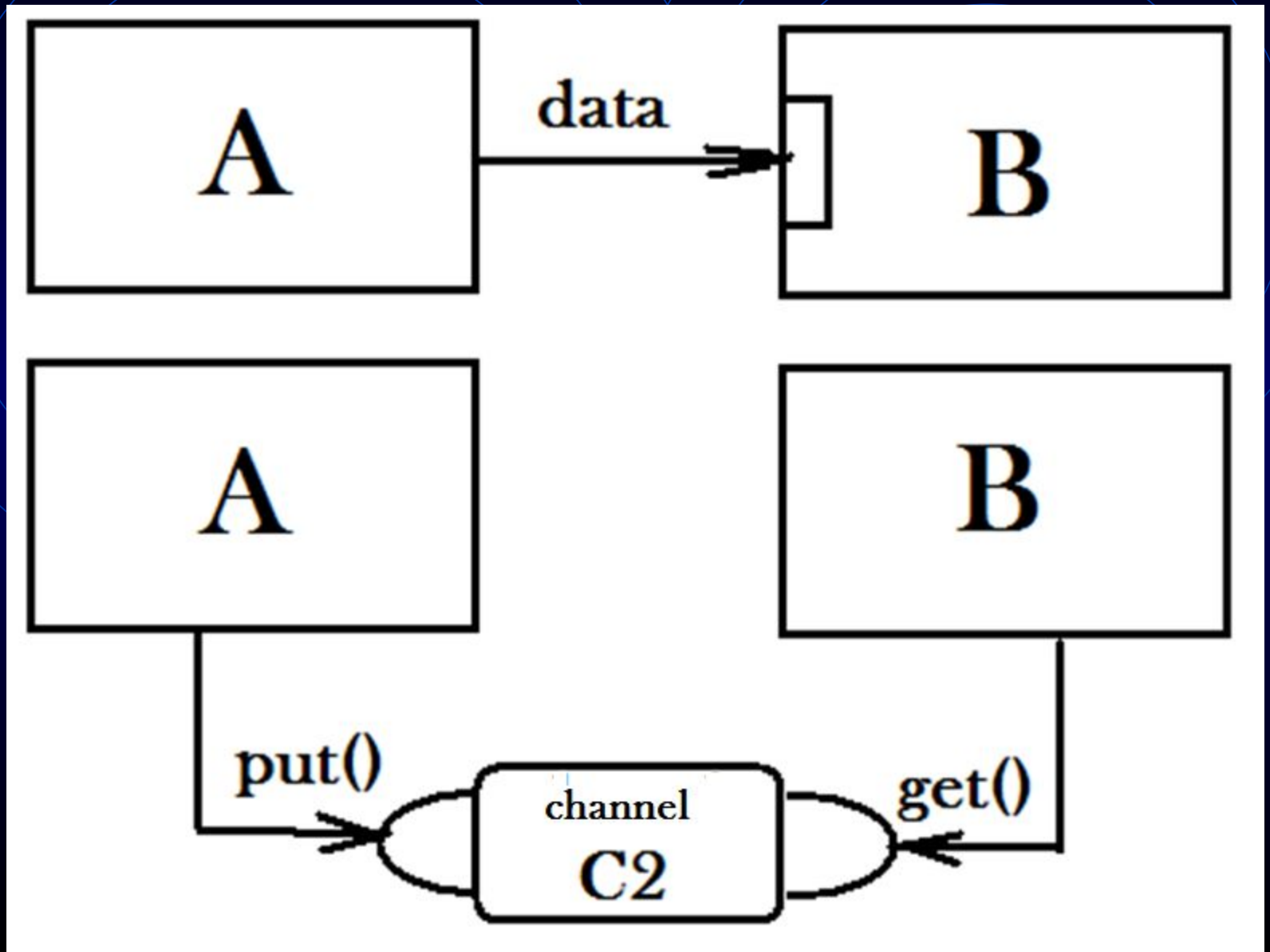
Семафоры

1) Открыть и создать : Open – Create
   Если уже существует, Create возвращает его handle
   и GetLastError() возвращает
   ERROR_ALREADY_EXISTS
2) Закрыть CloseHandle(obj);
3) DWORD WINAPI WaitForSingleObject(
   __in HANDLE *hHandle*,
   __in DWORD *dwMilliseconds*);
4) BOOL WINAPI ReleaseSemaphore(
   __in HANDLE *hSemaphore*,
   __in LONG *lReleaseCount*,
   __out LPLONG *lpPreviousCount*);

Семафоры



Каналы



«Потребитель - Производитель»

TSemaphore free = new TSemaphore(1);
TSemaphore empty = new TSemaphore(0);

процесс «Производитель»	**процесс «Потребитель»**

while (true){
<генерирует информацию>
free.P();
<запись данных в buf>
empty.V();
}
	while (true){
	empty.P();
	<Забирает данные из buf>
	free.V();
	<Обработка данныех>
	}

Пример “потребителя” с критической секцией на двух семафорах

```
for (;;)      {
    WaitForSingleObject(emptySem, INFINITE);
                    // забираем данные:
    memcpy(&Data, Buffer, sizeof(Data));
    ReleaseSemaphore(freeSem, 1,  NULL);
      // что-то делаем вне критической секции:
    for (int i=0;i<n; i++){……}
}
```

Завершение по тайм-ауту

```
for (;;){
    DWORD WINAPI  result =
WaitForSingleObject  (emptySem,   // семафор
CONSUMER_SLEEP_TIME  );      // тайм-аут
    if (result  == WAIT_TIMEOUT)  break;
            // не дождались  реакции  другого
             //  процесса -  наша реакция
    memcpy(&Data, Buffer, sizeof(Data));
    ReleaseSemaphore ( freeSem, 1, NULL );
    // обрабатываем данные
  }
```

Класс двоичный семафор

```
class TSemapfore{
  private:
     HANDLE Sem;
  public:
     void P(){   WaitForSingleObject(Sem, INFINITE);  }
     void V(){   ReleaseSemaphore(Sem, 1, NULL);   }
     TSemapfore(const char * name, int startState ){
Sem = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
                                  true, (LPCWSTR)name);
    int s = (startState > 0);
    if (Sem  == NULL)    Sem  = CreateSemaphore (NULL,
                                        s, 1,(LPCWSTR)name);}
    ~TSemapfore(){}
};
```

Канал для потоков одного приложения

```
Class TChannel {
private:
    TSemaphore free;
        TSemaphore empty;
    TData data;  // здесь храним данные канала
public:
    void put(TData  t)  ;
        TData get(TData * resultData) ;
        TChannel (){
                free = new TSemaphore(1);
                empty = new TSemaphore(0);
        }
        TChannel () {}
};
```

Потребитель с фиксировонным p

```
DWORD WINAPI ConsumerThreadProc (PVOID p) {
   class TChannel *   channel  = new TChannel
("MyChannel ");
   ULONG ConsumerId = (ULONG)(ULONG_PTR)p;
    int index =  ConsumerId, sum = 0;
    while (index)    {
         // index = p– количество  порций для обработки
      sum += channel -> get( );
       index--;
   }
   return 0;
}
```

Потребитель «пока есть данные»

```
DWORD WINAPI ConsumerThreadProc (PVOID p) {
 …
while ( true )    {
      DWORD WINAPI  result = WaitForSingleObject(empty,
10000);
      if (result  == WAIT_TIMEOUT)   break;
      sum += data;
      ReleaseSemaphore(free, 1, NULL);
    }

    printf ("Consumer %u stop  sum = %d \n", ConsumerId, sum);
    return 0;
}
```

Производитель с фиксированным p

```
DWORD WINAPI ProducerThreadProc (PVOID p) {
    class TChannel*  channel = new TChannel
("MyChannel");
    ULONG ProducerId = (ULONG)(ULONG_PTR)p;
    int index =  ProducerId;
    while (index)    {
        channel  -> put( index ); // положили в канал
       index--;
    }
 return 0;
}
```

Main – создали два производителя на 1900 и 1100 порций записи

```
int main () {
    DWORD id;
    HANDLE hProducer1 = CreateThread (
        NULL, 0, ProducerThreadProc, (PVOID) 1900,
0, &id);
    HANDLE hProducer2 = CreateThread (
        NULL, 0, ProducerThreadProc, (PVOID) 1100,
0, &id);
```

Main – создали три потребителя

```
int main () {
int main () {
…
   HANDLE hConsumer1 = CreateThread (
      NULL, 0, ConsumerThreadProc, (PVOID) 1, 0, &id);

 HANDLE hConsumer2 = CreateThread (
      NULL, 0, ConsumerThreadProc, (PVOID) 2, 0, &id);

HANDLE hConsumer3 = CreateThread (
      NULL, 0, ConsumerThreadProc, (PVOID) 3, 0, &id);
```

Main – завершение работы

```
int main () {
…
 WaitForSingleObject (hProducer1, INFINITE);
   WaitForSingleObject (hProducer2, INFINITE);

   WaitForSingleObject (hConsumer1, INFINITE);
   WaitForSingleObject (hConsumer2, INFINITE);
   WaitForSingleObject (hConsumer3, INFINITE);

return 0;
}
```

Main – работа


C:\...istr_PSMP\04_Объекты_ядра\thread>mai
Producer 1100 started
Producer 1900 started
Consumer 1 started
Consumer 2 started
Consumer 3 started
Producer 1100 stop sum=1100
Producer 1900 stop sum=1900
Consumer 1 stop sum = 1060
Consumer 2 stop sum = 984
Consumer 3 stop sum = 956
C:\...istr_PSMP\04_Объекты_ядра\thread>
1Help 2UserMn 3View 4Edit 5Copy 6R

Канал для потоков одного приложения не работает для разных приложений

```
Class  TChannel {
private:
    TSemaphore free;
        TSemaphore empty;

|// ПРОБЛЕМЫ С ДОСТУПОМ
// К ДАННЫМ:
            TData data;
};
```

Файл, отображаемый на память

```
class  TChannel  {
private:
HANDLE semAvailable;
                    //  Семафор занятого канала
HANDLE semEmpty;
                    //  Семафор свободного канала
HANDLE fileMem;
                    //  Файл, отображаемый на память
void * buffer;        //  Буфер для записи - чтения
public:
  …
};
```

Файл, отображаемый на память

```
void * Buffer;
   //  Буфер для записи  - чтения данных

HANDLE FileMem;
     //  Файл, отображаемый на память
FileMem = CreateFileMapping(…);

    //  установили адрес на область файла:
Buffer=MapViewOfFile(FileMem…);
```

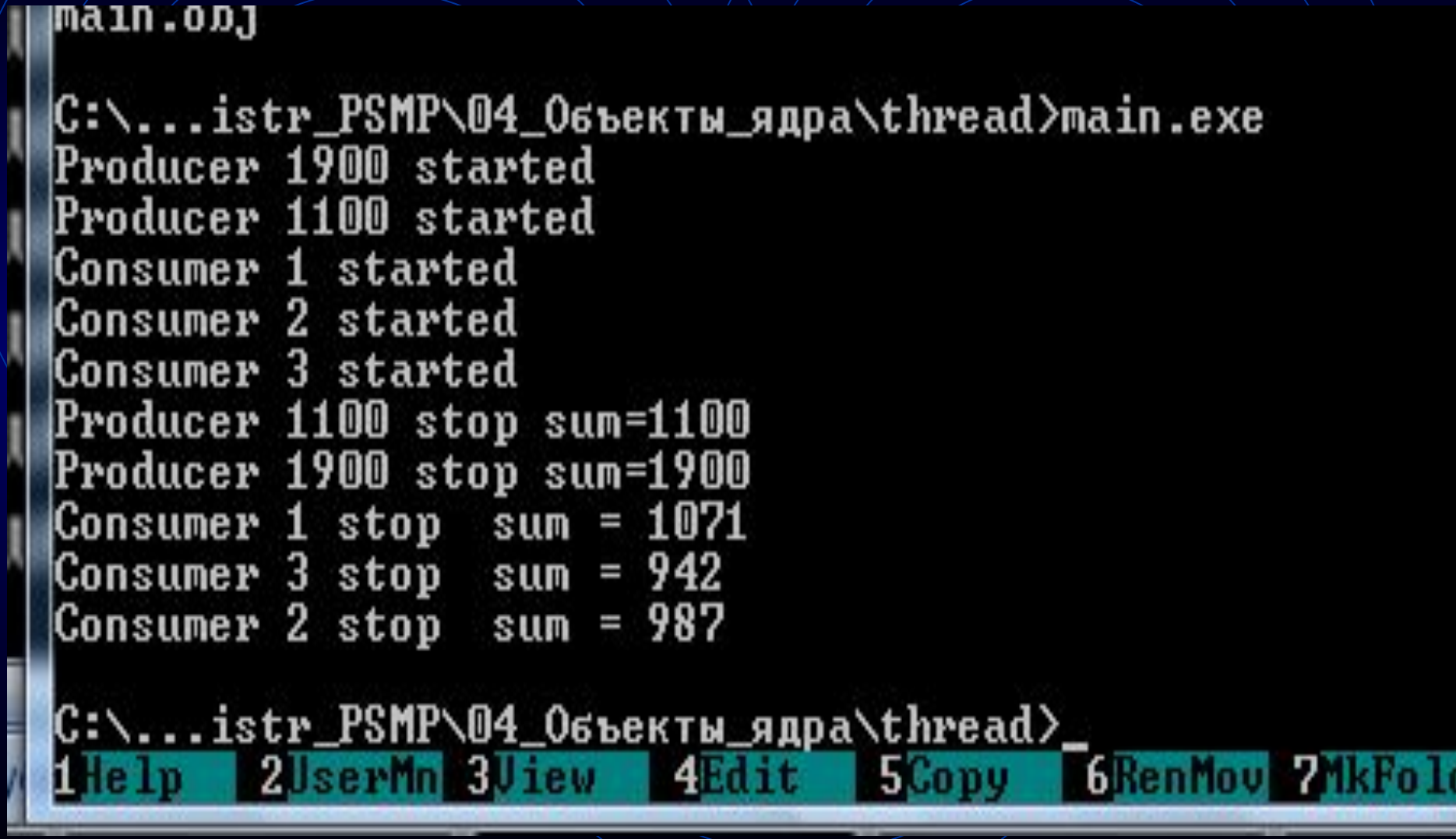
Файл, отображаемый на память

```
HANDLE FileMem;       //  Файл, отображаемый на память
FileMem=OpenFileMapping(
          FILE_MAP_ALL_ACCESS,
// все права на файл, кроме FILE_MAP_EXECUTE
          false,     //  handle  не наследуется при CreateProcess
         "MY_NAME");
if (FileMem==NULL)     FileMem = CreateFileMapping(
         (HANDLE)0xFFFFFFFF,
 //   INVALID_HANDLE_VALUE --- СОЗДАЕМ НОВЫЙ
         NULL,  // LPSECURITY_ATTRIBUTES
          PAGE_READWRITE,    //  вид доступа к данным
          0,4096,   // размер
          "MY_NAME");
```

Файл, отображаемый на память

```
void * Buffer;
  //  Буфер для записи  - чтения данных
if (FileMem!=NULL)
   Buffer=MapViewOfFile(
            FileMem,   // Handle файла
            FILE_MAP_ALL_ACCESS,
            0,    0,  // смещение
           4096);   // длина данных
else { printf("error: FILE_MAP \n");
       // Все плохо!!!!
 }
```

Файл, отображаемый на память



Семафоры и отображаемые на память файлы

Вывод

Работа с объектами ядра позволяет организовать передачу данных и любое другое межпроцессорное взаимодействие между любыми выполняющимися приложениями и между любыми потоками разных приложений.

Завершение процессов

- Один из потоков вызывает функцию ExitProcess(exitCode) (завершение процесса и всех его потоков) ☹ ☹ ☹

-Поток другого процесса вызывает функцию TerminateProcess(handle, exitCode) ☹ ☹ ☹

-Функция процесса возвращает управление ☹ или ☺

-Все потоки процесса завершаются сами ☺ ☺ ☺

- Завершение программы диспетчером задач ☹ ☹ ☹

Ожидание завершения всех дочерних процессов в родительском процессе

Управляющий поток должен каким-либо способом проинформировать рабочие потоки о том, что пора заканчивать работу (например, установив глобальный флаг), после чего дождаться, пока все потоки не завершатся, сделав все необходимые для корректного завершения действия: освободив ресурсы, информировав клиентов о завершении работы, закрыв сетевые соединения и т.п.

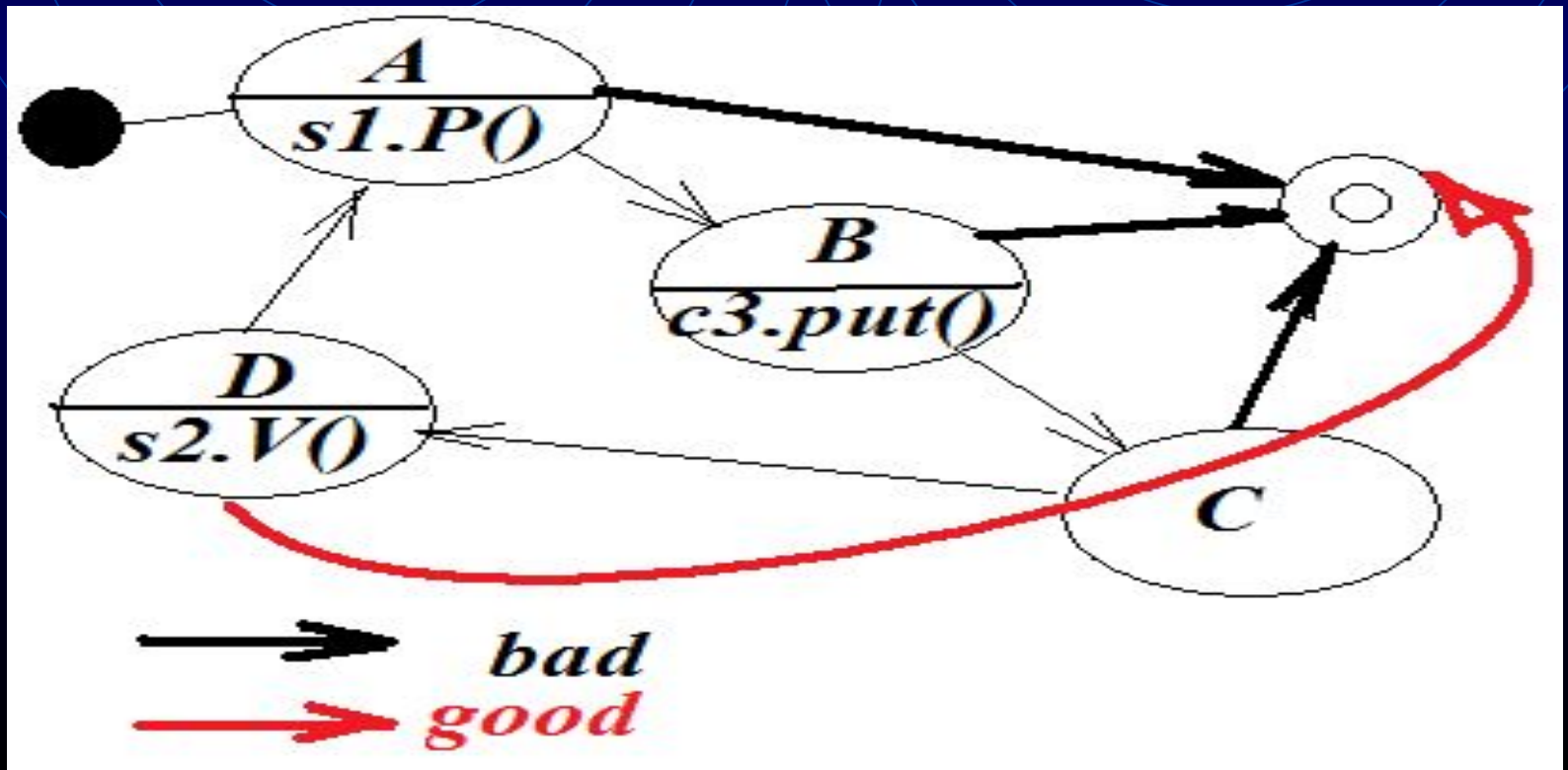
Ожидание завершения всех дочерних процессов в родительском процессе

```
PROCESS_INFORMATION pi;
DWORD exitCode;
if( !CreateProcess( …&pi){

   WaitforSingleObject(pi.hProcess, INFINITE);
GetExitCodeProcess(pi.hProcess, & exitCode);

CloseHandle(pi.hProcess);   // описатель процесса, а не процесс!!
}
```

Диаграмма состояний процесса

-Завершение процесса может привести к зависанию других связанных с данным процессов

-Переход в заключительное состояние - это завершение процесса



Wait - функцин

DWORD WINAPI WaitForSingleObject(__in HANDLE *hHandle*, __in DWORD *dwMilliseconds*);

HANDLE *hHandle – объект ядра, у которого проверяется состояние*

Возвращаемое значение == причина, почему процесс вновь стал активным:
WAIT_TIMEOUT
WAIT_OBJECT

……
}

Wait - функцин

DWORD WINAPI WaitForSingleObject(__in HANDLE *hHandle*, __in DWORD *dwMilliseconds*);

HANDLE *hHandle – объект ядра, у которого проверяется состояние*

Возвращаемое значение == причина, почему процесс вновь стал активным:
WAIT_TIMEOUT
WAIT_OBJECT

```
DWORD dw = WaitForSingleObject(m_hmtxQ, dwTimeout);
   if (dw == WAIT_OBJECT_0) {
      // Этот поток имел исключительные права на доступ
     // к данным
```

… …

Сложные Wait - функции

```
DWORD WINAPI SignalObjectAndWait(
    __in HANDLE hObjectToSignal,
    __in HANDLE hObjectToWaitOn,
    __in DWORD dwMilliseconds,
    __in BOOL bAlertable );
```

Перевести с свободное состояние один объект ядра и ждать другой объект ядра

bAlertable = true - функция в данном потоке возвращает управление и поток продолжает выполняться

Критические секции

- Семафор в Windows — это объект ядра., для его работы требуется переход из режима пользователя в режим ядра. Это дорогая операция, но она предоставляет мощный механизм синхронизации, который можно использовать через границы процессов. Если синхронизация необходима внутри единственного процесса., то способность
- Семафора работать через границы приводит к непроизводительным затратам. Для избавления от этих затратMicrosoft реализовала критическую секцию, обеспечивающую блокировку пользовательского уровня.

Критические секции

- void InitializeCriticalSection(LPCRITICALJECTION lpCS);
- void EnterCriticalSection(LPCRITICAL SECTION lpCS);
- BOOL TryEnterCriticalSection(LPCRITICALJECTION lpCS);
- void LeaveCriticalSection(LPCRITICALJECTION lpCS);
- void DeleteCriticalSection(LPCRITICAL SECTION lpCS);

Пулы потоков

Функция использования пула потоков,

```
BOOL QueueUserWorkItem (
        LPTHREAD_START__ROUTINE Function.
         PVOID Context.
         ULONG Flags );
```

Первый параметр— указатель на функцию, которую должен выполнять поток из пула. Эта функция должна иметь вид:

```
DWORD WINAPI Function( LPVOID parameter );
```

Параметр Flags = WTEXECUTELONGFUNCTION если все потоки заняты, то автоматически создается новый поток.

Мьютексы в C++ (11)

- mutex — нет контроля повторного захвата тем же потоком;
- recursive_mutex — повторные захваты тем же потоком допустимы, ведётся счётчик таких захватов;
- timed_mutex — нет контроля повторного захвата тем же потоком, поддерживается захват мьютекса с тайм-аутом;
- recursive_timed_mutex — повторные захваты тем же потоком допустимы, ведётся счётчик таких захватов, поддерживается захват мьютекса с тайм-аутом.

Лабораторная работа 5

Реализовать схему взаимодействия процессов на основе потоков – одно приложение и потоки соответствуют отдельным процессам.

Для каждого процесса системы реализовать отдельное приложение.

Протестировать нормальную работу системы после принудительного завершения одного или нескольких процессов.

Отладка многопоточных приложений

Сложность многопоточных программ приводит к множеству возможных состояний, в которых программа может находиться в любой момент времени:

- Мертвые блокировки приводят к тому, что приложение или вся система зависает.
- Появляются недетерминированные сбои,
- Ошибки неожиданно могут проявляться, причем невозможно повторить ситуацию, в которой возникла ошибка,
- Многопоточные ошибки могут не появляться при выполнении под управлением отладчика.

Отладка многопоточных приложений

Многопоточные приложения по своей сути более сложные, чем однопоточные. Главной причиной этого является большое количество крайних случаев, которые могут произойти, и широкий диапазон возможных путей выполнения приложения.

Подход «код пишется сейчас, а проектирование и тестирование откладываются на потом» при проектировании многопоточного приложения — это рецепт катастрофы

Отладка многопоточных приложений

При разработке многопоточных приложений:

- Используйте признанные паттерны параллельного программирования, безопасность которых подтверждена.
- Избегайте мертвых блокировок, захватывайте ресурсы согласованно (обедающие философы).
- По возможности разрабатывайте приложение так, чтобы оно могло выполняться последовательно.
- Регистрируйте сообщение до и после возникновения синхронизирующего события (журнал последовательности событий).
- Используйте окно потоков в отладчике Microsoft Visual Studio.

Лабораторная работа № 5

1. В соответствии со схемой лабораторной работы № 4 реализовать классы выбранных примитивов синхронизации на основе объектов ядра
2. Реализовать приложение, в котором каждый процесс – отдельный поток
3. Тело каждого потока - это бесконечный цикл , в котором выполняются действия в соответствии с диаграммой состояний
4. Перенести действия каждого процесса в отдельное приложение

ИТОГ: межпроцессорное взаимодействие потоков и приложений на одной платформе