Оглавление

лабораторная работа 4 синхронизация 1
Теория 1
Чему нужно научиться 13
Задание 13

ЛАБОРАТОРНАЯ РАБОТА 4 СИНХРОНИЗАЦИЯ

Теория

Многопоточность внутри одного процесса или нескольких основывается на способности процесса координировать свою работу с работой другого процесса. В этой лабораторной работе мы рассмотрим синхронизацию потоков внутри адресного пространства одного процесса (а дальше – синхронизацию различных процессов).
Все потоки процесса выполняются в его адресном пространстве и используют общие ресурсы для решения общей проблемы.
Синхронизирующий механизм позволяет нескольким потокам координировать работу. Существует два вида синхронизации: синхронная и асинхронная.
Пусть два потока 1 и 2 выполняются в одном процессе:
при синхронном сценарии (см. Рис. 1 Синхронные операции) поток 1 выполняет часть работы, затем поток 2 выполняет вторую часть работы, которая основана на результатах работы 1-ой части. А затем поток 1 выполняет 3-ю часть работы, которая требует завершения 2-ой части.
Этот сценарий требует, чтобы потоки были в состоянии точно координировать исполнение определенных частей работы.

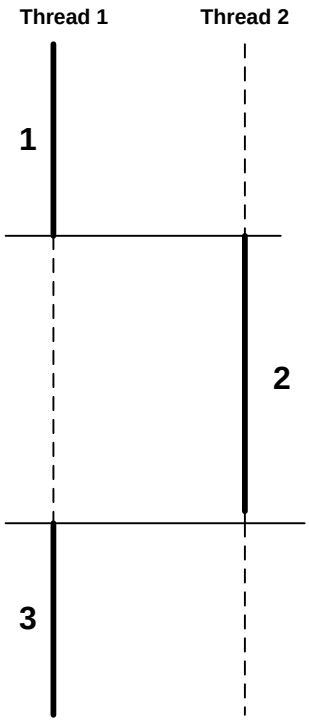

Рис. 1 Синхронные операции

В асинхронном сценарии (см. Рис. 2 Асинхронные операции) – потоки 1 и 2 работают с тремя частями: поток 1 заканчивает первую часть работы, а затем поток 2 заканчивает
2-ю часть параллельно с потоком 1, который будет заканчивать 3-ю часть.

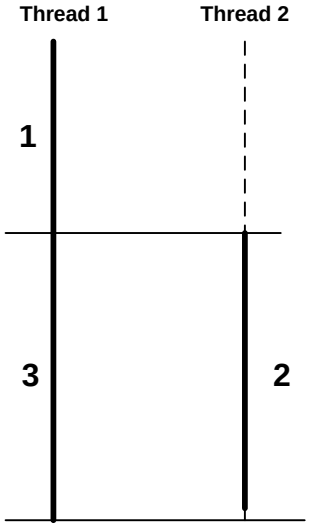

Рис. 2 Асинхронные операции

Две части (2 и 3) выполняются одновременно, если в системе не один процессор.

Потоки 1 и 2 предполагается выполнять параллельно в одном адресном пространстве. Все переменные, описанные в программе процесса, являются разделяемыми (кроме автоматических, которые определяются в стеке потока). Переменные сегмента данных – статические переменные в С программе - разделяются (используются совместно) потоками процесса. Если два или более потока получают доступ к разделенным переменным, выполняя критическую секцию (и по крайней мере хотя бы один задает значение переменной) – будут проблемы. Решение проблемы – использовать механизм на уровне ОС для взаимного исключения. Если один поток находится в критической секции, тогда другие потоки не входят в критическую секцию до тех пор, пока первый там находится.
Для синхронизации потоков для координации и взаимного исключения выполняется мы будем использовать синхронизирующие объекты (Executive Obfects):

- Event – событие
- Mutex – мутант
- Semaphore – семафор
- Timer – таймер.

Поведение всех этих синхронизационных объектов аналогично семафорам Dijkstra:
P(s): [while (s = = 0) wait( ); s-- ;]
V(s): [s++;]
Где P и V атомарные операции.
Wait позволяет другим потокам выполняться пока s=0. Каждый объект NT (dispather object), который строится на базе объекта ядра, содержит переменную состояния, которая позволяет объекту находиться в состоянии свободен (signaled) или занят (nonsignaled). Например, когда процесс или поток заканчивается, его объект (dispather object) переходит в свободное (signaled) состояние. Различные операции приводят к изменению состояния объекта. Программно можно проверять состояние объекта, используя функции ожидания ***WaitForSingleObject*** и ***WaitForMultipleObject***.
Процесс и поток – эти объекты создаются в занятом состоянии и остаются в нем всю свою жизнь. Когда процесс или поток заканчиваются, используя ***ExitProcess*** или ***ExitThread***, их состояние меняется, они переходят из занятого состояния в свободное.

```
ThrdHandle = CreateThread(….);

WaitForSingleObject(thrdHandle, INFINITE);
CloseHandle(thrdHandle);
```

Когда поток создался, он начинает существовать в занятом состоянии. Когда поток заканчивается, он меняет свое состояние на свободное. Тем временем контролирующий процесс блокирует себя используя ***WaitForSingleObject*** до тех пор, пока не закончится поток (child).

Схема при исполнении таймера такая же. Он создается в занятом состоянии. Поток ждет истечения промежутка времени, используя ***WaitForSingleObject*** с описателем таймера до тех пор, пока он не изменит состояние на свободное.

Функции ожидания

Функции ожидания ***WaitForSingleObject*** и используются с любыми объектами синхронизации.

```
DWORD WaitForSingleObject(
        HANDLE hObject,
        DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, hObject, идентифицирует объект, который может находиться в состоянии занят/свободен. Второй параметр, dwMilliseconds, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта. Если мы задаем INFINITE, то это значит, что вызывающий поток готов ждать этого события хоть целую вечность. Правда, передача INFINITE не всегда безопасна. Если объект так и не перейдет в свободное состояние, то вызывающий поток никогда не проснется; правда процессорное время он при этом тратить не будет.

```
WaitForSingleObject(hProcess, INFINITE);
```

Рассмотрим еще один пример:

```
DWORD dw = WaitForSingleObject( hProcess, 5000);
switch( dw )
{
        case WAIT_OBJECT_0:
// процесс завершается
break;
         case WAIT_TIMEOUT:
                // процесс не завершился в течение 5000 мс
                break;
        case WAIT_FAILED:
                // неправильный вызов функции
                break;
}
```

Вызывающий поток не получит процессорное время, пока не завершится указанный процесс или не пройдет 5000 мс (смотря что случится раньше).
Значение, возвращаемое функцией ***WaitForSingleObject***, указывает, почему вызывающий поток снова стал планируемым. Если объект освободился, то функция вернет WAIT_OBJECT_0, а если истекло время таймаута, то

WAIT_TIMEOUT. Если передан неверный параметр, функция возвращает – WAIT_FAILED (для выяснения конкретной причины вызовите ***GetLastError***). Функция ***WaitForMultipleObject*** работает аналогично, но позволяет ждать освобождения сразу нескольких объектов или какого-то одного из заданного списка объектов:

```
DWORD WaitForMultipleObjects(
        DWORD dwCount,
        CONST HANDLE* phObjects,
        BOOL fWaitAll,
DWORD dwMilliseconds);
```

Параметр dwCount задает количество объектов ядра (его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS=64). Параметр phObjects – это указатель на массив описателей объектов ядра.
WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр fWaitAll и задает этот ваш выбор. Если он равен TRUE, то функция не даст потоку возобновить работу пока не освободятся все объекты.
Параметр dwMilliseconds идентичен одноименному параметру функции ***WaitForSingleObject***.
Возвращаемое функцией значение сообщает, почему возобновилось выполнение вызвавшего ее потока. Если вы задали fWaitAll TRUE, то функция вернет WAIT_OBJECT_0, если все объекты перешли в свободное состояние. Если же fWaitAll FALSE, то функция вернет управление, как только один из объектов(любой) перейдет в свободное состояние. Чтобы узнать какой именно объект освободился, необходимо вычесть из возвращаемого значения WAIT_OBJECT_0.

```
HANDLE h[3];
h[0] = hPracess1;
h[1] = hPracess2;
h[2] = hPracess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch( dw )
{
        case WAIT_FAILED:
                // неправильный вызов
                break;

        case WAIT_TIMEOUT:
                // ни один из объектов не освободился в течение 5000 мс
                break;

        case WAIT_OBJECT_0 + 0:
```

```
                // завершился процесс h[0]
                break;

case WAIT_OBJECT_0 + 1:
                // завершился процесс h[1]
                break;

case WAIT_OBJECT_0 + 2:
                // завершился процесс h[2]
                break;
}

#define  N….
.....
HANDLE    thrdHandle[N;]
…….
for(i=0; i<N;i++)
{
        thrdHandle[i] = Create Thread(…);
}
…….
WaitForMultipleObjects(N, thrdHandle,TRUE,INFINITE);
```

Что случится если dwMilliseconds=0 при вызове WaitForSingleObject или WaitForMultipleObjects? (т.е. таймаут немедленно истекает при вызове).

События

Объекты события используется для того, чтобы сообщить о том, что что-то произошло в данном потоке, другим потокам.

```
HANDDE CreateEvent(
        PSECURITY_ATTRIBUTES psa,
        BOOL fManualReset,
        BOOL fInitialState,
PCTSTR pszName);
```

Параметр fManualReset (булева переменная) сообщает системе, какого типа событие вы хотите создать:

- TRUE – со сбросом вручную;
- FALSE – с авто сбросом.

Параметр fInitialState определяет начальное состояние события:

- TRUE – свободное;
- FALSE – занятое.

После того как система создает объект событие, ***CreateEvent*** возвращает описатель события, который помещается в таблицу описателей данного

процесса. Ненужный больше объект событие следует, как всегда закрывать, используя ***CloseHandle***.
Создав событие вы можете управлять его состоянием, используя ***SetEvent*** и ***ResetEvent.***

BOOL ***SetEvent***(HANDLE hEvent);
BOOL ***ResetEvent***(HANDLE hEvent);

Для событий с авто сбросом действует определенное правило. Когда ожидающий его поток наконец то его дождался, объект событие автоматически сбрасывается в занятое состояние (отсюда и произошло название).
Объекты события используется для того, чтобы сообщить о том, что что-то произошло в данном потоке, другим потокам. Можно использовать auto-reset или manual-reset. Повторим, что auto-reset позволяет только одному потоку понять, что объект событие перешел в состояние свободен, а затем он немедленно переходит обратно в состояние занят. Использование manual-reset дает понять многим потокам, что событие произошло, но объекты события должны быть перегружены используя ***SetEvent.***
Используем события для реализации сценария на Рис. 1 Синхронные операции.

```
int main ( )
{
        // this is threadX
        ……..
        evntHandle[ PART2 ] = CreateEvent( NULL, TRUE, FALSE, NULL );
        evntHandle[ PART3 ] = CreateEvent( NULL, TRUE, FALSE, NULL );
        ……..
        CreateThread(…, threadY,…);
        // Do part 1
        ……
        SetEvent( evntHandle[ PART2 ] );
        WaitForSingleObject( evntHandle[ PART3 ], INFINITE );
        ResetEvent( evntHandle[ PART3 ] );
        // Do part 3
        ……..
}

DWORD WINAPI threadY ( LPVOID )
{
        ……
        WaitForSingleObject( evntHandle[ PART2 ], INFINITE );
        ResetEvent( evntHandle[ PART2 ] );
        // Do part 2
```

```
        …….
        SetEvent( evntHandle[ PART3 ] );
        …….
}
```

Мьютексы

Объект мьютекс (mutex) также существует исключительно для синхронизации, хотя он специально был создан для управления критическими секциями. Объект мьютекс может принадлежать потоку владельцу или он может быть без владельца. Владение означает, что поток держит мьютекс. Так если мьютекс в состоянии занят, то владеющий поток в критической секции, защищаемой им.

Поток может стать владельцем мьютекса при его создании, получив его описатель (open) или используя функцию ожидания. Он может освободить мьютекс (перевести его в состояние свободен), используя функцию ***ReleaseMutex***.

```
HANDLE CreateMutex(
                LPSECURITY ATTRIBUTES lpMutexAttributes,
                // указать на атрибут безопасности
                BOOL bInitialOwner, // флаг начального собственника
                LPCTSTR IpName     // указатель на имя
                                );
```

Атрибут bInitialOwner определяет будет ли вызывающий поток владельцем мьютекса.

Если bInitialOwner = TRUE (и вызов функции успешен), то мьютекс будет создан в состоянии занят и вызывающий поток будет его владельцем. Подобно другим функциям для создания объектов ***CreateMutex*** вернет ошибку, если использовать имя, которое уже используется (***GetLastError*** вернет ERROR ALREADY EXISTS).

После того, как мьютекс создан, любой поток процесса может его использовать. Если поток не является владельцем мьютекса, но хочет им стать, он должен использовать функцию ожидания. Подобно функции ожидания для объектов других типов возврат из функции ожидания произойдет, если объект перейдет в состояние свободен. Успешное ожидание (вы можете проверить код возврата) позволяет вызываемому потоку стать владельцем, а состояние объекта изменится на занятое. ***ReleaseMutex*** освобождает мьютекс и его состояние меняется на свободное.

Объекты мьютексы можно использовать для решения проблем критических секций. Предположим потоки X и Y разделяют ресурс R. Оба потока выполняют вычисления, обращаются к R и затем выполняют вычисления снова. Так как R разделяемый ресурс доступ к нему это критическая секция.

```
int main( )
{
        //This is controlling thread
```

```
        ……..
        // Open resource R
        //Create the Mutex Object with no owner  ( signaled )
        ……Create Thread(…, workerThrd,…); // Create thread X
        ……Create Thread(…, workerThrd,…); // Create thread Y
        …….
}

DWORD WINAPI workerThrd( LPVOID )
{
        …….
        while(….)
        {
                // выполнить работу
                …….
                //получить mutex
                while (WaitForSingleObject(mutexR) != WAIT_OBJECT_0);
                //Доступ к ресурсу R
                ReleaseMutex(mutexR);
        }
        …..
}
```

Семафоры

```
HANDLE CreateSemaphore(
        PSECURITY_ATTRIBUTES psa,
        LONG lInitialCount,
        LONG lMaximumCount,
        PCTSTR pszName);
```

Параметр lMaximumCount сообщает системе максимальное число ресурсов. Параметр lInitialCount указывает, сколько из этих ресурсов доступно изначально.
Объект имеет внутреннюю переменную, принимающую значение от 0 до lMaximumCount (должен быть >0). Когда объект создается, начальное значение внутренней переменной может быть установлено любым в этом промежутке и задается lInitialCount. Состояние объекта семафора определяется значением внутренней переменной.

```
HANDLE hSem = CreateSemaphore(NULL, 0, 5, NULL);
```

Внутреннее значение объекта семафора может изменять, используя функции. Функция ожидания уменьшает значение внутренней переменной. Функция *ReleaseSemaphore* увеличивает значение внутренней переменной (счетчика).

BOOL R***eleaseSemaphore*** (
HANDLE hSemafhore, //описатель объекта семафора
LONG lReleaseCount, //число + к текущему значению
LPLONG lPreviousCount //указатель на предыдущие значения
);

lReleaseCount – число, которое прибавляется к значению семафора. lPreviousCount – указатель на переменную, которая должна содержать значение, которое было до вызова ***ReleaseSemaphore*** (может быть NULL, если вы не заботитесь о предыдущем значении).
Когда использовать семафоры?
Предположим потоки X и Y оба используют ресурс R (который состоит из кусочков), каждый может потребовать К- кусочков и использовать их некоторое время, а затем вернуть.

```
#define N……
int main()
{
	//это контролирующий поток
	…….
	//создать объект семафор
	SemaphoreR = CreateSemaphore( NULL, 0, N, NULL );
	…..CreateThread(…..,WorkerThrd,…);  //создать поток X
	…..CreateThread(…..,WorkerThrd,…);  //создать поток Y
	.....
}

DWORD WINAPI WorkerThrd(LPVO1D)
{
	while(….)
	{
		//выполнить некоторую работу
		…..
		for (i=0; i<k; i++
		while( WaitForSingleObject(Semaphore) != WAIT_OBJECT_O )
		……
		//освободить к – кусочков
		Release Semaphore (SemaphoreR,K,NULL);
		……
	}
}
```

Таймеры

HANDLE ***CreateWaitableTimer***(

```
    PSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    PCTSTR pszName);
```

Параметр fManualReset задает тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со взбросом вручную, возобновляется выполнение всех ожидающих его потоков, а когда в свободное состояние переходит таймер с авто сбросом – лишь одного из потоков.
Объект ожидаемый таймер всегда создается в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию ***SetWaitableTimer*** (см. MSDN).

Критические секции

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
DWORD WINAPI  FirstThread(PVOID pvParam)
{
    while(g_nIndex < MAX_TIMES)
    {
g_dwTimes[g_nIndex] = GetTickCount();
g_nIndex++;
    }
    return 0;
}

DWORD WINAPI  SecondThread(PVOID pvParam)
{
    while(g_nIndex < MAX_TIMES)
    {
g_nIndex++;
g_dwTimes[g_nIndex - 1] = GetTickCount();
    }
    return 0;
}
```

Если бы обе функции выполнялись независимо, то каждая заполняла бы массив g_dwTimes набором возрастающих чисел. Но в нашем случае обе функции будут одновременно обращаться к глобальному массиву g_dwTimes. Допустим пусть первым начал выполняться поток, выполняющий SecondThread , и значение счетчика g_nIndex увеличилось до 1. А дальше система вытеснила ее поток и начала исполнять FirstThread, а та заносит в g_dwTimes[1] системное время. После этого процессор вновь переключается на выполнение второго потока и уже тот присваивает

элементу g_dwTimes[1-1] значение системного времени. Так как эта операция выполняется позже, то значение системного времени, записанное в 0-ой элемент массива, будет больше чем в первый.
Теперь попробуем исправить этот фрагмент, используя критическую секцию.

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;

DWORD WINAPI  FirstThread(PVOID pvParam)
{
        for( BOOL fContinue = TRUE; fContinue; )
        {
                EnterCriticalSection(&g_cs);
if (g_nIndex < MAX_TIMES)
                {
g_dwTimes[g_nIndex] = GetTickCount();
g_nIndex++;
                }
                else fContinue = FALSE;
                LeaveCriticalSection(&g_cs);
        }
        return 0;
}

DWORD WINAPI  SecondThread(PVOID pvParam)
{
        for( BOOL fContinue = TRUE; fContinue; )
        {
                EnterCriticalSection(&g_cs);
if (g_nIndex < MAX_TIMES)
                {
g_nIndex++;
g_dwTimes[g_nIndex-1] = GetTickCount();
                }
                else fContinue = FALSE;
                LeaveCriticalSection(&g_cs);
        }
        return 0;
}
```

Необходимо было создать экземпляр структуры данных CRITICAL_SECTION – g_cs, а затем вставить вызов EnterCriticalSection

перед началом работы с разделяемыми ресурсами (у нас это g_nIndex и g_dwTimes) и вызов LeaveCriticalSection после окончания работы с ними.
Если у вас есть ресурсы, которыми разные потоки пользуются вместе, вы можете поместить их в критическую секцию и она будет их защищать. Вызов EnterCriticalSection позволяет выяснить свободна или занята критическая секция. EnterCriticalSection допускает вход потока в критическую секцию, если определяет, что та свободна. В противном случае (критическая секция занята) эта функция заставит поток ждать пока она не освободится.
Закончив работу с критическим ресурсом, поток должен вызвать функцию LeaveCriticalSection. Таким образом, он уведомляет систему о том, что критическая секция освободилась. Если вы забудите это сделать, то система будет считать, что ресурс все еще занят и не позволит обратиться к нему другим ждущим потокам.
Элементы структуры CRITICAL_SECTION необходимо инициализировать до обращения какого-либо потока к защищенному ресурсу. Структура инициализируется вызовом:

VOID ***InitializeCriticalSection***(PCRITICAL_SECTION pcs);

InitializeCriticalSection должна быть вызвана до того, как один из потоков обратится к EnterCriticalSection. Если критическая секция никому больше не нужна, удалите ее вызвав:

VOID ***DeleteCriticalSection***(PCRITICAL_SECTION pcs);

Нельзя удалять критическую секцию, если она еще нужна какому-нибудь потоку.

Чему нужно научиться

Необходимо научиться синхронизировать работу потоков процесса.

Задание

Уровень 1 (А)

Использовать любой синхронизационный объект для синхронизации нескольких (например, двух) потоков в одном процессе. В качестве разделяемого ресурса использовать стандартный ввод/вывод. Каждый из потоков должен осуществлять прием строк от пользователя и отвечать пользователю (общаться с пользователем). Причем возможность общения должна быть предоставлена только одному потоку. Передать возможность общения, другому потоку нужно вводя **next**, а завершать поток, вводя **exit.** Основной (базовый) поток процесса должен закончить работу только в том случае, когда пользователь завершит работу всех вспомогательных потоков.

Уровень 1 (B)

Создать два потока, которые выполняются в одном адресном пространстве (в одном процессе):

- поток – производитель;
- и поток – потребитель.

Массив buf содержит производимые и потребляемые данные (данные совместного использования (массив структур). Производитель заполняет массив, а потребитель извлекает данные из массива.

Код должен удовлетворять трем требованиям:

- потребитель не должен пытаться извлечь данные из буфера, если буфер пуст;
- производитель не должен пытаться поместить данные в буфер, если буфер полон;
- все поля структуры элемента массива должны быть заполнены.

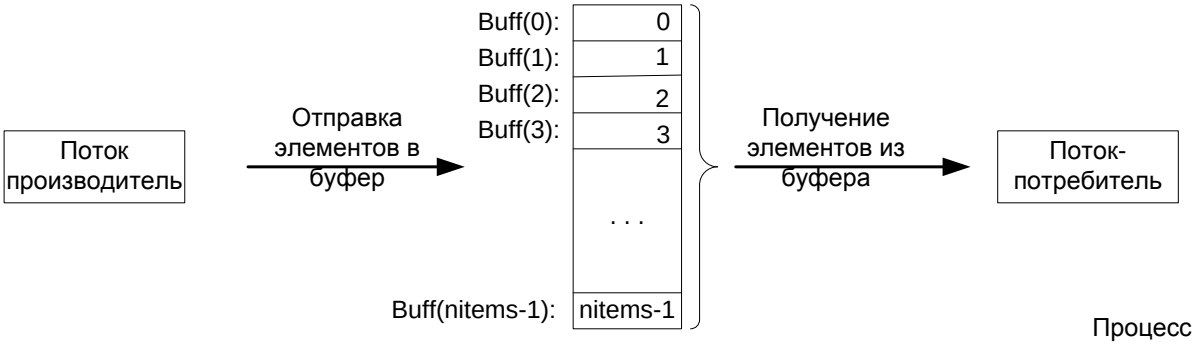

Рис. 3 Производитель и потребитель

Уровень 2 (B)

Пусть имеется несколько производителей и несколько потребителей.

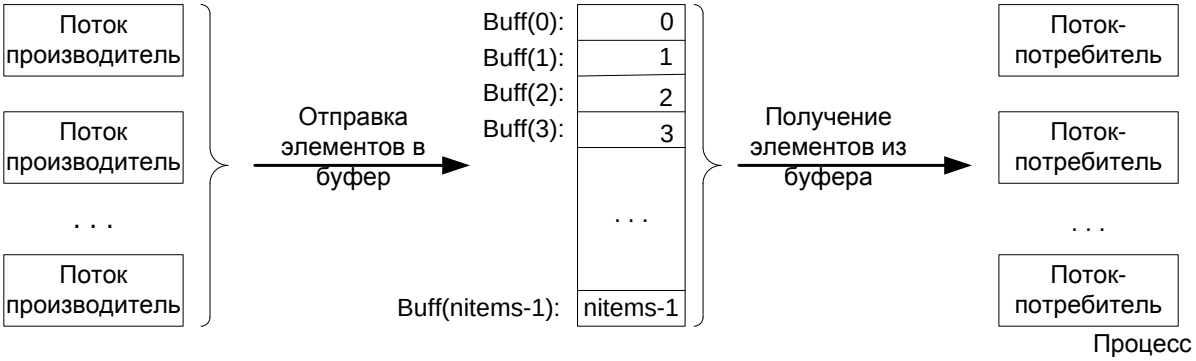

Рис. 4 Производители и потребители

Уровень 3

Придумать любую задачу на синхронизацию потоков процесса и реализовать ее.

Скелет кода для задания А

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#include <conio.h>

DWORD WINAPI Thread1(LPVOID lpParameter)
{
	…
	printf("Thread1 begins\n");

	while(!bExit)
	{
		// вставить ***WaitForSingleObject***

		gets(str);
		printf("User input %s\n", str);

		while((strcmp( str,"next"))&&(strcmp(str,"exit")))
		{
			gets(str);
			printf("User input %s\n", str);
		}

		if( !strcmp(str,"exit") )	…

		printf("Thread1 release control\n");

		// использовать ***SetEvent***
	} // while

	printf( "Thread1 finished\n" );
	…….
	return 0;
}

DWORD WINAPI Thread2(LPVOID lpParameter)
{
	…….
}

int main( int argc, char* argv[] )
{
	printf( "I am main! Hello!\n" );
```

```
        // создать объекты для синхронизации hNextEvent = CreateEvent(…);

        CreateThread(…Thread1,….);
        …….
        return 0;
}
```

Скелет кода 1 (В)

```
#define N…..
Event mutex = 1;
Semaphore full = 0;
Semaphore empty = N;

int main( )
{
        …….
        //создать производителя и потребителя
        ….. CreateThread(….., Producer,….);
        ….. CreateThread(….., Consumer,…);
        ……
}

Producer( )
{
        while( TRUE )
        {
                P( empty );
                P( mutex );
                        produce
                V( mutex );
                V( full );
        }
}

Comsumer( )
{
        while( TRUE )
        {
                P( full );
                P( mutex );
                        consume
                V( mutex );
                V(empty);
```

```
        }
    }
```