Дисциплина «Системы реального времени»
Лабораторная работа № 4

Тема: «Синхронизация потоков при помощи семафоров и критических секций».

Критические секции в Windows.

В операционных системах Windows проблема взаимного исключения для параллельных потоков, выполняемых в контексте одного процесса, решается при помощи объекта типа CRITICAL_SECTION, который не является объектом ядра операционной системы. Для работы с объектами этого типа используются следующие функции:

```
VOID   InitializeCriticalSection   (LPCRITICAL_SECTION lpCriticalSection);
VOID   EnterCriticalSection        (LPCRITICAL_SECTION lpCriticalSection);
BOOL   TryEnterCriticalSection     (LPCRITICAL_SECTION lpCriticalSection);
VOID   LeaveCriticalSection        (LPCRITICAL_SECTION  lpCriticalSection);
VOID   DeleteCriticalSection       (LPCRITICAL_SECTION  lpCriticalSection);
```

каждая из которых имеет единственный параметр, указатель на объект типа CRITICAL_SECTION. Все эти функции, за исключением TryEnterCriticalSection, не возвращают значения. Отметим, что функция TryEnterCriticalSection поддерживается только операционной системой Windows 2000.

Кратко рассмотрим порядок работы с этими функциями. Для этого предположим, что при проектировании программы мы выделили некоторый разделяемый ресурс и критические секции в параллельных потоках, которые имеют доступ к этому разделяемому ресурсу. Тогда для обеспечения корректной работы с этим ресурсом нужно выполнить следующую последовательность действий:

- определить в нашей программе объект типа CRITICAL_SECTION, имя которого логически связано с выделенным разделяемым ресурсом;
- проинициализировать объектом типа CRITICAL_SECTION при помощи функции *InitializeCriticalSection*;
- в каждом из параллельных потоков пред входом в критическую секцию вызвать функцию *EnterCriticalSection*, которая исключает одновременный вход в критические секции, связанные с нашим разделяемым ресурсом, для параллельно выполняющихся потоков;
- после завершения работы с разделяемым ресурсом, поток должен покинуть свою критическую секцию, что выполняется посредством вызова функции *LeaveCriticalSection*;
- после окончания работы с объектом типа CRITICAL_SECTION, необходимо освободить все системные ресурсы, которые использовались этим объектом. Для этой цели служит функция *DeleteCriticalSection*.

Теперь покажем работу этих функций на примере. Для этого сначала рассмотрим пример, в котором выполняются не синхронизированные параллельные потоки, а затем синхронизируем их работу, используя критические секции.

Программа

4.1. // Пример работы не синхронизированных потоков

```
#include <windows.h>
#include <iostream>

using namespace std;

DWORD WINAPI thread(LPVOID)
{
        int i,j;
```

```
	for (j = 0; j < 10; j++)
	{
		for (i = 0; i < 10; i++)
		{
			cout << j << ' ';
			cout  <<  flush;
			Sleep(22);
		}
		cout << endl;
	}

	return 0;
}

int main()
{
	int i,j;
	HANDLE		hThread;
	DWORD		IDThread;

	hThread=CreateThread(NULL, 0, thread, NULL, 0,
	&IDThread); if (hThread == NULL)
		return GetLastError();

	// так как потоки не синхронизированы,
	// то выводимые строки
	непредсказуемы for (j = 10; j < 20; j++)
	{
		for (i = 0; i < 10; i++)
		{
			cout << j << ' ';
			cout  <<  flush;
			Sleep(22);
		}
		cout << endl;
	}
	// ждем, пока поток thread закончит свою работу
	WaitForSingleObject(hThread, INFINITE);

	return 0;
}
```

В этой программе каждый из потоков main и thread выводит строки одинаковых чисел. Но из-за параллельной работы потоков, каждая выведенная строка может содержать не равные между собой элементы. Наша задача будет заключаться в следующем: нужно так синхронизировать потоки main и thread, чтобы в каждой строке выводились только равные между собой элементы. Следующая программа показывает решение этой задачи с помощью объекта типа CRITICAL_SECTION.

Программа 4.2.

```
// Пример работы синхронизированных потоков

#include <windows.h>
#include <iostream>

using namespace std;

CRITICAL_SECTION cs;

DWORD WINAPI thread(LPVOID)
{
	int i,j;

	for (j = 0; j < 10; j++)
```

```
	{
		// входим в критическую секцию
		EnterCriticalSection   (&cs);
		for (i = 0; i < 10; i++)
		{
			cout << j << '
			'; cout.flush();
		}
		cout << endl;
		// выходим из критической секции
		LeaveCriticalSection(&cs);
	}

	return 0;
}

int main()
{
	int i,j;
	HANDLE		hThread;
	DWORD		IDThread;
	// инициализируем критическую секцию
	InitializeCriticalSection(&cs);

	hThread=CreateThread(NULL, 0, thread, NULL, 0,
	&IDThread); if (hThread == NULL)
		return GetLastError();

	// потоки синхронизированы, поэтому каждая
	// строка содержит только одинаковые
	числа for (j = 10; j < 20; j++)
	{
		// входим в критическую секцию
		EnterCriticalSection(&cs);
		for (i = 0; i < 10; i++)
		{
			cout << j << '
			'; cout.flush();
		}
		cout << endl;
		// выходим из критической секции
		LeaveCriticalSection(&cs);
	}
	// закрываем критическую секцию
	DeleteCriticalSection(&cs);
	// ждем, пока поток thread закончит свою работу
	WaitForSingleObject(hThread, INFINITE);

	return 0;
}
```

Теперь рассмотрим использование функции TryEnterCriticalSection. Для этого просто заменим в приведенной программе вызовы функции EnterCriticalSection на вызовы функции TryEnterCriticalSection и будем отмечать успешные входы потоков в свои критические секции. Еще раз подчеркнем, что функция TryEnterCriticalSection работает только на платформе операционной системы Windows 2000.

Прогамма 4.3.

```
// Пример работы синхронизированных потоков.
// Работает только в Windows 2000.

#include <windows.h>
#include <iostream>

using namespace std;
```

```
CRITICAL_SECTION cs;

DWORD WINAPI thread(LPVOID)
{
	int i,j;

	for (j = 0; j < 10; j++)
	{
		// попытка войти в критическую секцию
		TryEnterCriticalSection
		(&cs); for (i = 0; i < 10; i++)
		{
			cout << j << " ";
			cout.flush();
		}
		cout << endl;
		// выход из критической секции
		LeaveCriticalSection(&cs);
	}

	return 0;
}

int main()
{
	int i, j;
	HANDLE		hThread;
	DWORD		IDThread;
	// инициализируем критическую секцию
	InitializeCriticalSection(&cs);

	hThread=CreateThread(NULL, 0, thread, NULL, 0,
	&IDThread); if (hThread == NULL)
		return GetLastError();

	// потоки синхронизированы, поэтому каждая
	// строка содержит только одинаковые
	числа for (j = 10; j < 20; j++)
	{
		// попытка войти в критическую секцию
		TryEnterCriticalSection(&cs);
		for (i = 0; i < 10; i++)
		{
			cout << j << " ";
			cout.flush();
		}
		cout << endl;
		// выход из критической секции
		LeaveCriticalSection(&cs);
	}
	// удаляем критическую секцию
	DeleteCriticalSection(&cs);
	// ждем завершения работы потока thread
	WaitForSingleObject(hThread, INFINITE);

	return 0;
}
```

Отметим, что, так как объекты типа CRITICAL_SECTION не являются объектами ядра операционной системы, то работа с ними происходит несколько быстрее, чем с объектами ядра операционной системы, так как в этом случае программа меньше обращается к ядру операционной системы.

Семафоры Дийкстры.

Семафор – это неотрицательная целая переменная, значение которой может изменяться только при помощи неделимых операций . Под понятием неделимая операция мы понимаем такую операцию , выполнение которой не может быть прервано. Семафор считается свободным, если его значение больше нуля, в противном случае семафор считается занятым. Пусть s – семафор, тогда над ним можно определить следующие неделимые операции:

```
P(s) {
        если s >0 то s = s – 1;                    // поток продолжает работу
        иначе ждать освобождения s;        // поток переходит в состояние ожидания
}

V(s) {
        если потоки ждут освобождения s, то освободить один поток;
        иначе s = s + 1;
}
```

Семафор с операциями P и V называется семафором Дийкстры, который первым использовал семафоры для решения задач синхронизации. Из определения операций над семафором видно, что если поток выдает операцию P и значение семафора больше нуля, то значение семафора уменьшается на 1 и этот поток продолжает свою работу, в противном случае поток переходит в состояние ожидания до освобождения семафора другим потоком. Вывести из состояния ожидания поток, который ждет освобождения семафора, может только другой поток, который выдает операцию V над этим же семафором. Потоки, ждущие освобождения семафора, выстраиваются в очередь к этому семафору. Дисциплина обслуживания очереди зависит от конкретной реализации. Очередь может обслуживаться как по правилу FIFO, так и при помощи более сложных алгоритмов, учитывая приоритеты потоков.

Семафор, который может принимать только значения 0 или 1, называется двоичным или бинарным семафором. Чтобы подчеркнуть отличие бинарного семафора от не бинарного семафора, то есть такого семафора, значение которого может быть больше 1, последний обычно называют считающими семафором. Покажем, как бинарный семафор может использоваться для моделирования критических секций и событий. Для этого сначала рассмотрим следующие потоки.

```
semaphor s = 1; // семафор свободен

void thread_1( )
{
                .
                .
                .
        P(s);
                if (n%2 == 0)
                        n = a;
                else
                        n = b;
        V(s);
                .
                .
                .
}
```

```
void thread_2( )
{
                .
                .
                .
        P(s);
                        n++;
        V(s);
                .
                .
                .
}
```

Как следует из определения операций над семафором, данный подход решает проблему взаимного исключения одновременного доступа к переменной n для потоков thread_1 и thread_2. Таким образом, бинарный семафор позволяет решить проблему взаимного исключения.

Теперь предположим, что поток thread_1 должен производить проверку значения переменной n только после того, как поток thread_2 увеличит значение этой переменной. Для решения этой задачи модифицируем наши программы следующим образом:

```
semaphor s = 0; // семафор занят

void thread_1( )                          void thread_2( )
{                                         {
            .                                     .
            .                                     .
            .                                     .
      P(s);                                       n++;
            if (n%2 == 0)                         V(s);
                  n = a;                          .
            else                                  .
                  n = b;                          .
            .                             }
            .
            .
}
```

Как видно из этих программ, бинарный семафор позволяет также решить задачу условной синхронизации.

Семафоры в Windows.

Семафоры в операционных системах Windows описываются объектами ядра Semaphores, Семафор находится в сигнальном состоянии , если его значение больше нуля. В противном случае семафор находится в не сигнальном состоянии. Создаются семафоры посредством вызова функции CreateSemaphore, которая имеет следующий прототип:

```
HANDLE CreateSemaphore(
      LPSECURITY_ATTRIBUTES   lpSemaphoreAttribute,   // атрибуты защиты
      LONG          lInitialCount,          // начальное значение семафора
      LONG          lMaximumCount,          // максимальное значение семафора
      LPCTSTR       lpName                  // имя семафора
);
```

Как и обычно, пока значение параметра lpSemaphoreAttributes будем устанавливать в NULL. Основную смысловую нагрузку в этой функции несут второй и третий параметры. Значение параметра lInitialCount устанавливает начальное значение семафора, которое должно быть не меньше 0 и не больше его максимального значения, которое устанавливается параметром lMaximumCount.

В случае успешного завершения функция CreateSemaphore возвращает дескриптор семафора, в случае неудачи – значение NULL. Если семафор с заданным именем уже существует, то функция CreateSemaphor возвращает дескриптор этого семафора, а функция GetLastError, вызванная после функции CreateSemaphor вернет значение ERROR_ALREADY_EXISTS.

Значение семафора уменьшается на 1 при его использовании в функции ожидания. Увеличить значение семафора можно посредством вызова функции ReleaseSemaphore, которая имеет следующий прототип:

```
BOOL ReleaseSemaphore(
      HANDLE        hSemaphore,             // дескриптор семафора
      LONG          lReleaseCount,          // положительное число,
                                            // на которое увеличивается значение семафора
      LPLONG        lpPreviousCount         // предыдущее значение семафора
);
```

В случае успешного завершения функция ReleaseSemaphore возвращает значение TRUE, в случае неудачи – FALSE. Если значение семафора плюс значение параметра lReleaseCount

больше максимального значения семафора, то функция ReleaseSemaphore возвращает значение FALSE и значение семафора не изменяется.

Значение параметра lpPreviousCount этой функции может быть равно NULL. В этом случае предыдущее значение семафора не возвращается.

Приведем пример программы, в которой считающий семафор используется для синхронизации работы потоков. Для этого сначала рассмотрим не синхронизированный вариант этой программы.

Программа 4.4.

```
// Несинхронизированные потоки

#include <windows.h>
#include <iostream>

using namespace std;

volatile int a[10];

DWORD WINAPI thread(LPVOID)
{
	int i;

	for (i = 0; i < 10; i++)
	{
		a[i] = i + 1;
		Sleep(17);
	}

	return 0;
}

int main()
{
	int i;
	HANDLE	hThread;
	DWORD	IDThread;

	cout << "An initial state of the array:
	"; for (i = 0; i < 10; i++)
		cout << a[i] <<'
	'; cout << endl;

	// создаем поток, который готовит элементы массива
	hThread = CreateThread(NULL, 0, thread, NULL, 0,
	&IDThread); if (hThread == NULL)
		return GetLastError();

	// поток main выводит элементы
	массива cout << "A modified state of the
	array: "; for (i = 0; i < 10; i++)
	{
		cout << a[i] << '
		'; cout.flush();
		Sleep(17);
	}
	cout << endl;

	CloseHandle(hThread);

	return 0;
}
```

Теперь кратко опишем работу этой программы. Поток thread последовательно присваивает элементам массива «a» значения, которые на единицу больше чем их индекс.

Поток main последовательно выводит элементы массива «a» на консоль. Так как потоки thread и main не синхронизированы, то неизвестно, какое состояние массива на консоль поток main. Наша задача состоит в том, чтобы поток main выводил на консоль элементы массива «a» сразу после их подготовки потоком thread. Для этого мы используем считающий семафор. Следующая программа показывает, как этот считающий семафор используется для синхронизации работы потоков.

Программа 4.5.

```
// Пример синхронизации потоков с использованием семафора

#include <windows.h>
#include <iostream>

using namespace std;

volatile int a[10];
HANDLE hSemaphore;

DWORD WINAPI thread(LPVOID)
{
	int i;

	for (i = 0; i < 10; i++)
	{
		a[i] = i + 1;
		// отмечаем, что один элемент готов
		ReleaseSemaphore(hSemaphore,1,NULL);
		Sleep(500);
	}

	return 0;
}

int main()
{
	int i;
	HANDLE	hThread;
	DWORD	IDThread;

	cout << "An initial state of the array:
	"; for (i = 0; i < 10; i++)
		cout << a[i] <<'
	'; cout << endl;
	// создаем семафор
	hSemaphore=CreateSemaphore(NULL, 0, 10,
	NULL); if (hSemaphore == NULL)
		return GetLastError();

	// создаем поток, который готовит элементы массива
	hThread = CreateThread(NULL, 0, thread, NULL, 0,
	&IDThread); if (hThread == NULL)
		return GetLastError();

	// поток main выводит элементы массива
	// только после их подготовки потоком
	thread cout << "A final state of the array: ";
	for (i = 0; i < 10; i++)
	{
		WaitForSingleObject(hSemaphore,
		INFINITE); cout << a[i] << ' ';
		cout.flush();
	}
	cout << endl;
```

```
    CloseHandle(hSemaphore);
    CloseHandle(hThread);

    return 0;
}
```

Может возникнуть следующий вопрос: почему для решения этой задачи используется именно считающий семафор и почему его максимальное значение равно 10. Конечно, поставленную задачу можно было бы решить и другими способами. Но дело в том, что считающие семафоры предназначены именно для решения подобных задач. Подробнее, считающие семафоры используются для синхронизации доступа к однотипным ресурсам, которые производятся некоторым потоком или несколькими потоками, а потребляются другим потоком или несколькими потоками. В этом случае значение считающего семафора равно количеству произведенных ресурсов, а его максимальное значение устанавливается равным максимально возможному количеству таких ресурсов. При производстве единицы ресурса значение семафора увеличивается на единицу, а при потреблении единицы ресурса значение семафора уменьшается на единицу. В нашем примере ресурсами являются элементы массива , заполненные потоком thread, который является производителем этих ресурсов . В свою очередь поток main является потребителем этих ресурсов, которые он выводит на консоль. Так как в общем случае мы не можем сделать предположений о скоростях работы параллельных потоков, то максимальное значение считающего семафора должно быть установлено в максимальное количество производимых ресурсов. Если поток потребитель ресурсов работает быстрее чем поток производитель ресурсов, то, вызвав функцию ожидания считающего семафора, он вынужден будет ждать, пока поток-производитель не произведет очередной ресурс. Если же наоборот, поток-производитель работает быстрее чем поток-потребитель, то первый поток произведет все ресурсы и закончит свою работу, не ожидая, пока второй поток потребит их. Такая синхронизация потоков производителей и потребителей обеспечивает их максимально быструю работу.

Доступ к существующему семафору можно открыть с помощью одной из функций CreateSemaphore или OpenSemaphore. Если для этой цели используется функция CreateSemaphore, то значения параметров lInitialCount и lMaximalCount этой функции игнорируются, так как они уже установлены другим потоком, а поток, вызвавший эту функцию, получает полный доступ к семафору с именем, заданным параметром lpName. Теперь рассмотрим функцию OpenSemaphore, которая используется в случае, если известно, что семафор с заданным именем уже существует. Эта функция имеет следующий прототип:

```
HANDLE OpenSemaphore(
    DWORD       dwDesiredAccess,    // флаги доступа
    BOOL        bInheritHandle,     // режим наследования
    LPCTSTR     lpName              // имя события
);
```

Параметр dwDesiredAccess определяет доступ к семафору, и может быть равен любой логической комбинации следующих флагов:

```
SEMAPHORE_ALL_ACCESS
SEMAPHORE_MODIFY_STATE
SYNCHRONIZE
```

Флаг SEMAPHORE_ALL_ACCESS устанавливает для потока полный доступ к семафору. Это означает, что поток может выполнять над семафором любые действия. Флаг SEMAPHORE_MODIFY_STATE означает, что поток может использовать только функцию ReleaseSemaphore для изменения значения семафора. Флаг SYNCHRONIZE означает, что поток может использовать семафор только в функциях ожидания. Отметим, что последний режим поддерживается только на платформе Windows NT/2000.

Задача.

Написать программу для консольного процесса, который состоит из трёх потоков: **main** , **work, и третьего** (см. варианты).

Глобальные переменные не использовать!

Индивидуальные варианты:

4.1 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;
ввести число k;
запустить поток **work**;
запустить поток **SumElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;
известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будут выведены на консоль k элементов массива).

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу программы используя критическую секцию для** синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Найти в массиве неповторяющиеся элементы (разместить их в массиве слева, остальные соответственно справа). Элементы - символы.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком **main,** использовать **бинарный семафор!***):

ждёт от потока **main** сигнал о начале суммирования;
выполнить суммирование элементов итогового массива до заданной позиции k; вывести итоговую сумму.

4.2 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;
запустить поток **work**;
запустить поток **SumElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Найти в массиве повторяющиеся элементы (разместить их группы в массиве слева, остальные соответственно справа). Элементы - вещественные числа.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;
известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будет сформирован итоговый массив.

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком work, использовать критическую секцию!*):

ждёт от потока **work** сигнал о начале суммирования;
выполнить суммирование элементов итогового массива; вывести итоговую сумму.

4.3 Поток **main** должен выполнить следующие действия:
создать массив, размерность и элементы которого вводятся пользователем с консоли;
вывести размерность и элементы исходного массива на консоль;
ввести число k;
запустить поток **work**;
запустить поток **SumElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;
известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будут выведены на консоль k элементов).

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу программы используя критическую секцию для** синхронизации с потоком **main, объяснить отличия, если есть!**)*:
запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Сортировка методом "пузырька". Элементы - вещественные числа двойной точности.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком main**, использовать бинарный семафор!***)**:
ждёт от потока **main** сигнал о начале суммирования;
выполнить суммирование элементов итогового массива до заданной позиции k; вывести итоговую сумму.

4.4 Поток **main** должен выполнить следующие действия:
создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;
ввести число k;
запустить поток **work**;
запустить поток **MultElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:
запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Поиск в массиве элементов из диапазона [A,B] (разместить их в массиве слева, остальные элементы массива - заполнить нулями). Элементы - целые числа без знака. Числа A,B ввести в потоке main.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;
известить поток **MultElement** о начале работы (момент запуска произойдёт после того, будет сформирована часть итогового массива (когда будут найдены все элементы из диапазона [A, B]).

Поток **MultElement** должен выполнить следующие действия (*Для синхронизации с потоком work**, использовать критическую секцию!***)**:

ждёт от потока **work** сигнал о начале работы;
выполнить произведение элементов итогового массива (когда будут найдены все элементы из диапазона [A, B]);
вывести произведение.

4.5 Поток **main** должен выполнить следующие действия:
создать массив, размерность и элементы которого вводятся пользователем с консоли;

вывести размерность и элементы исходного массива на консоль; ввести число k;
запустить поток **work**;
запустить поток **SumElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;
известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будут выведены на консоль k элементов массива).

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу программы используя критическую секцию для** синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Сортировка выбором. Элементы - символы.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком main, использовать бинарный семафор!*):

ждёт от потока **main** сигнал о начале суммирования;
выполнить суммирование элементов (кодов символов) итогового массива до заданной позиции k;
вывести итоговую сумму.

4.6 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;
ввести число k;
запустить поток **work**;
запустить поток **SumElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива. Запросить число A.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Поиск в массиве элементов >A (разместить их в массиве слева, остальные элементы массива - заполнить нулями). Элементы - целые числа без знака. Число A ввести в потоке main.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;
известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будет сформирован итоговый массив.

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком work, использовать критическую секцию!*):

ждёт от потока **work** сигнал о начале суммирования;
выполнить суммирование элементов итогового массива; вывести итоговую сумму.

4.7 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;
ввести число k;
запустить поток **work**;
запустить поток **SumElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.

выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;
известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будут готовы к печати k - элементов массива).

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу программы используя критическую секцию для** синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Поиск в массиве простых чисел (разместить их в массиве слева, остальные элементы массива - справа). Элементы - целые числа без знака.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком main, использовать бинарный семафор!*):

ждёт от потока **main** сигнал о начале суммирования;
выполнить суммирование элементов итогового массива до заданной позиции k; вывести итоговую сумму.

4.8 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли;
вывести размерность и элементы исходного массива на консоль; запустить поток **work**;
запустить поток **CountElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
Запросить символ X.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Поиск в массиве элементов =X (разместить их в массиве слева, остальные элементы массива - справа). Элементы - символы. X ввести в потоке main.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;
известить поток **CountElement** о начале работы (момент запуска произойдёт после того, будет сформирован итоговый массив.

Поток **CountElement** должен выполнить следующие действия (*Для синхронизации с потоком work, использовать критическую секцию!*):

ждёт от потока **work** сигнал о начале суммирования;
подсчитать количество элементов равных X;
вывести итоговую сумму.

4.9 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;
ввести число k;
запустить поток **work**;
запустить поток **MultElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;
известить поток **MultElement** о начале работы (момент запуска произойдёт после того, будут выведены на консоль k элементов массива).

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу программы используя критическую секцию для** синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;

Поиск в массиве элементов <A (разместить их в массиве слева, остальные элементы массива - справа). Элементы - вещественные числа. Число A ввести в потоке main.

извещать поток main о новом элементе;

после каждого готового элемента отдыхать в течение заданного интервала времени;

Поток **MultElement** должен выполнить следующие действия (*Для синхронизации с потоком main, использовать бинарный семафор!*):

ждёт от потока **main** сигнал о начале суммирования;

выполнить произведение элементов итогового массива до заданной позиции k; вывести итоговое произведенеи.

4.10 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;

запустить поток **work**;

запустить поток **SumElement**;

освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.

выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;

Поиск в массиве лексем, (разделители – цифры). Полученные лексемы поместить в массиве слева, разделитель - пробел, остальные элементы - заполнить символом ‘0’. Элементы массива - символы.

извещать поток main о новом элементе;

после каждого готового элемента отдыхать в течение заданного интервала времени;

известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будет сформирован итоговый массив.

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком work, использовать критическую секцию!*):

ждёт от потока **work** сигнал о начале суммирования;

выполнить суммирование элементов (кодов) итогового массива; вывести итоговую сумму.

4.11 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;

ввести число k;

запустить поток **work**;

запустить поток **SumElement**;

освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.

выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будут выведены на консоль k элементов массива).

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;

Приведение массива к палиндрому (получившейся палиндром поместить в массиве слева, а лишние элементы соответственно – справа) Элементы - символы

извещать поток main о новом элементе;

после каждого готового элемента отдыхать в течение заданного интервала времени;

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком main, использовать критическую секцию!*):

ждёт от потока **main** сигнал о начале суммирования;
выполнить суммирование элементов (кодов) итогового массива до заданной позиции k;
вывести итоговую сумму.

4.12 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;
запустить поток **work**;
запустить поток **MultElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу программы используя критическую секцию для** синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Сортировка выбором. Элементы - целые числа.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;
известить поток **MultElement** о начале работы (момент запуска произойдёт после того, будет сформирован итоговый массив.

Поток **MultElement** должен выполнить следующие действия (*Для синхронизации с потоком work, использовать бинарный семафор!*):

ждёт от потока **work** сообщения о начале суммирования;
выполнить произведение элементов итогового массива; вывести произведение.

4.13 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли;
вывести размерность и элементы исходного массива на консоль;
ввести число k;
запустить поток **work**;
запустить поток **SumElement**;
освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.
выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;
известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будут выведены на консоль k элементов массива).

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу программы используя критическую секцию для** синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;
Поиск в массиве элементов, соответствующих цифрам (слева поместить в массив цифры, а остальные элементы массива - заполнить пробелами). Элементы - символы.
извещать поток main о новом элементе;
после каждого готового элемента отдыхать в течение заданного интервала времени;

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком main, использовать бинарный семафор!*):

ждёт от потока **main** сообщения о начале суммирования;
выполнить суммирование элементов (кодов) итогового массива до заданной позиции k; вывести итоговую сумму.

4.14 Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;

запустить поток **work**;

запустить поток **SumElement**;

освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.

выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;

Поиск в массиве лексем, начинающихся с цифры (разделители – пробел и тире). Полученные лексемы поместить в массиве слева, а лишние элементы - -заполнить символом подчеркивания: «_»). Элементы - символы.

извещать поток main о новом элементе;

после каждого готового элемента отдыхать в течение заданного интервала времени;

известить поток **SumElement** о начале суммирования (момент запуска произойдёт после того, будет сформирован итоговый массив.

Поток **SumElement** должен выполнить следующие действия (*Для синхронизации с потоком work, использовать критическую секцию!*):

ждёт от потока **work** сообщения о начале суммирования;

выполнить суммирование элементов (кодов) итогового массива; вывести итоговую сумму.

*4.15*Поток **main** должен выполнить следующие действия:

создать массив, размерность и элементы которого вводятся пользователем с консоли; вывести размерность и элементы исходного массива на консоль;

запустить поток **work**;

запустить поток **Sum/CountElement**;

освобождение выходной поток stdout после вывода на консоль каждого нового элемента массива.

выводить на экран поэлементно элементы массива (итогового) параллельно с работой потока **work**;

Поток **work** должен выполнить следующие действия *(Для синхронизации с потоком **main** – использовать **семафор. Проверить работу используя бинарный семафор** для синхронизации с потоком **main, объяснить отличия, если есть!**)*:

запросить у пользователя временной интервал, требуемый для отдыха после подготовки одного элемента в массиве;

Поиск в массиве лексем, начинающихся с цифры (разделители – пробел и тире). Полученные лексемы поместить в массиве слева, а лишние элементы - -заполнить символом подчеркивания: «_»). Элементы - символы.

извещать поток main о новом элементе;

после каждого готового элемента отдыхать в течение заданного интервала времени;

известить поток **Sum/CountElement** о начале суммирования (момент запуска произойдёт после того, будет сформирован итоговый массив.

Поток **Sum/CountElement** должен выполнить следующие действия (*Для синхронизации с потоком work, использовать критическую секцию!*):

ждёт от потока **work** сообщения о начале суммирования;

выполнить суммирование и подсчёт элементов (до символов подчеркивания: «_») итогового массива,; вывести результаты.