Министерство образования и науки Украины
Севастопольский национальный технический университет

Методические указания

к лабораторным работам по дисциплине
«Параллельные и распределенные вычисления»
для студентов дневной и заочной форм обучения
направления 0.915 – «Компьютерная инженерия»

Севастополь

2006

УДК 519.6

Методические указания к лабораторным работам по дисциплине «Параллельные и распределенные вычисления» для студентов дневной и заочной форм обучения направления 0.915 – «Компьютерная инженерия» / Сост. Т.А. Абрамов, Г.Г.Сергеев. – Севастополь: Изд-во СевНТУ, 2006. – 43с.

Целью методических указаний является обеспечение лабораторных работ по дисциплине «Параллельные и распределенные вычисления».

Методические указания рассмотрены и утверждены на заседании кафедры КиВТ (протокол № 4 от 19 декабря 2005 г.).

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент: Овчинников А.Л. – старший преподаватель кафедры информационных систем СевНТУ.

СОДЕРЖАНИЕ

.

1. ЛАБОРАТОРНАЯ РАБОТА №1. ИЗУЧЕНИЕ СРЕДСТВ БЛОКИРОВКИ ФАЙЛОВ В MS DOS

1.1. Цель работы:

Изучить методы синхронизации и взаимодействия процессов в мультипрограммных ОС на примере средств, предоставляемых MS DOS для блокировки одновременного доступа к файлам.

1.2. Введение

Сетевые (Netware) и мультипрограммные операционные системы (Windows, Unix и др.) обеспечивают возможность параллельной обработки данных. В данной лабораторной работе предлагается изучить возможности совместной обработки файлов на базе средств, предоставляемых ОС Windows в режиме эмуляции ОС MS DOS.

1.3. Методика выполнения работы

1) Изучить методические указания к работе.
2) Создать в среде Borland Pascal (Borland C) программу, обеспечивающую выполнение операций над файлом прямого доступа, описанную в разделе «Пример программы, выполняющей операции над файлом».
3) Изменить программу в соответствии с вариантом задания к лабораторной работе, добавить средства блокировки.

1.4. Варианты заданий

Вариант задания (таблица 1.1) выбирается по формуле (N mod 24)+1, где N – последние две цифры зачетной книжки студента.

Таблица 1.1 – Варианты заданий к лабораторной работе №1

№ вар.	Язык Программ.	Тип данных	Режим блокировки
1	Pascal	Byte	Блокировка файла
2	Pascal	Byte	Блокировка записи
3	C	Byte	Блокировка файла
4	C	Byte	Блокировка записи
5	Pascal	Word	Блокировка файла
6	Pascal	Word	Блокировка записи
7	C	Word	Блокировка файла

8	C	Word	Блокировка записи

Продолжение таблицы 1.1

9	Pascal	Long	Блокировка файла
10	Pascal	Long	Блокировка записи
11	C	Long	Блокировка файла
12	C	Long	Блокировка записи
13	Pascal	Char	Блокировка файла
14	Pascal	Char	Блокировка записи
15	C	Char	Блокировка файла
16	C	Char	Блокировка записи
17	Pascal	Real	Блокировка файла
18	Pascal	Real	Блокировка записи
19	C	Real	Блокировка файла
20	C	Real	Блокировка записи
21	Pascal	Float	Блокировка файла
22	Pascal	Float	Блокировка записи
23	C	Float	Блокировка файла
24	C	Float	Блокировка записи

1.5. Содержание отчета

Постановка задачи. Описание программы. Методика и результаты испытаний. Выводы.

1.6. Контрольные вопросы

1. Понятие файла. Сформулируйте понятия дескриптор файла, файловая переменная, запись, смещение.
2. Совместная обработка файлов. Режимы открытия файла.
3. Функции MS DOS. Правила работы с функцией 5C.
4. Понятие критического ресурса и критической области.
5. Постановка задачи взаимного исключения.

1.7. Рекомендации по выполнению работы

1.7.1. Совместная обработка файлов

Операционная система MS DOS относится к однопрограммным операционным системам. Как следствие, в обычном режиме работы одновременный доступ к одному и тому же файлу невозможен (так как

невозможен одновременный запуск двух и более программ). Однако обрабатываемый файл может находиться не на локальной машине, а в сети – на файловом сервере. В этом случае файл может обрабатываться одновременно несколькими программами.

1.7.2. Типичные ошибки при совместном использовании файлов

Совместное использование одного и того же файла разными программами возможно при строгом соблюдении определенных правил. Рассмотрим простой пример – пусть файл содержит информацию о наличии некоторых товаров на складе (запись с номером N - количество товара под номером N). Предположим также, что с этим файлом работают два оператора – каждый на своем рабочем месте и работа каждого оператора сводится к следующим действиям:

1) принять заказ от покупателя;
2) проверить остаток заказанного товара;
3) уменьшить соответствующее количество в файле.

Теперь представим следующую ситуацию: остаток некоторого товара - 5 единиц. Два клиента хотят заказать 4 и 3 единицы этого товара соответственно. Первый оператор выполняет действия 1) и 2), после чего сервер переключается на обслуживание второго оператора. Второй оператор в свою очередь выполняет действия 1) и 2). Очередь доходит до первого оператора, который уменьшает количество на 4 (новый остаток = 1) и отправляет клиента за товаром. Второй оператор уменьшает количество на 3 (новый остаток = 2). В результате в файл попадает неверное значение (описание реакции покупателей оставим за рамками нашего примера).

Описанная проблема решается просто – необходимо исключить возможность вмешательства других программ во время выполнения одной из программ действий 2) и 3). В рассмотренной ситуации файл представляет собой критический ресурс (совместно используемый объект), а действия над ним в рамках одной программы составляют критическую область. Другими словами, мы имеем дело с задачей взаимного исключения.

1.7.3. Постановка задачи взаимного исключения.

Необходимо согласовать работу n>=2 параллельных процессов при использовании некоторого критического ресурса таким образом, чтобы удовлетворить следующим требованиям:

- одновременно внутри критической области должно находиться не более одного процесса;
- критические области не должны иметь приоритеты в отношении друг друга;
- остановка какого-либо процесса вне его критической области не должна влиять на дальнейшую работу процессов по использованию критического ресурса;
- решение о вхождении процессов в их критические области при одинаковом времени поступления запросов на такое вхождение и равноприоритетности процессов не откладывается на неопределенный срок, а является конечным во времени;
- относительные скорости развития процессов неизвестны и произвольны;
- любой процесс может переходить в любое состояние, отличное от активного, вне пределов своей критической области;
- освобождение критического ресурса и выход из критической области должны быть произведены процессом, использующим критический ресурс, за конечное время.

1.7.4. Функция 5C MS DOS и ее параметры

Функция 5C позволяет заблокировать определенную область файла. Попытка заблокировать уже заблокированную область файла приводит к ошибке.

AH=5C - номер функции.
AL=0 (1) - действие (0 – lock, 1-unlock).
BH=handle -системный номер файла (из файловой переменной).
CX:DX=offset - CX*65536+DX = смещение области.
SI:DI=length - SI*65536+DI = размер области.
AX - код ошибки, если CF<>0.

Для корректного использования функции 5C необходимо установить режим совместного доступа к файлу. В системе Borland Pascal для этого служит переменная filemode. Стандартное ее значение (02) соответствует режиму монопольного использования, значение $42 – режиму совместного доступа. В режиме монопольного использования попытка открытия уже открытого файла приведет к ошибке (файл блокируется целиком). В режиме совместного использования операционная система не ограничивает доступ к файлу, поэтому необходимо блокировать отдельные области файла, доступ к которым требуется исключать.

1.7.5. Пример программы, выполняющей операции над файлом

Необходимо составить программу, обеспечивающую чтение данных из файла прямого доступа и запись данных в файл. При выполнении операции записи должны запрашиваться позиция (номер записи) и данные, при выполнении операции чтения запрашивается позиция, данные выводятся на экран.

```
Program lab0;
Var f: file of integer;
    d,p,i: integer;
begin
  filemode:=$42;  assign(f,'data.dat');
{$I-}
  reset(f);
{$I+}
  if ioresult<>0 then begin
    rewrite(f); close(f);
    reset(f);
  end;
  repeat
    writeln('Введите код действия (0-выход, 1-
запись, 2-чтение');
    readln(i);
    case i of
    1: begin
         writeln('Введите номер записи');
         readln(p); seek(f,p);
         writeln('Введите данные');
         readln(d); write(f,d);
         writeln('Данные записаны');
         readln;
       end;
    2: begin
         writeln('Введите номер записи');
         readln(p); seek(f,p);
         read(f,d);
         writeln('Данные: ',d);
```



```
         readln;
       end;
  until i=0;
  close(f);
end.
```

2. ЛАБОРАТОРНАЯ РАБОТА №2. ИЗУЧЕНИЕ СРЕДСТВ СОВМЕСТНОЙ ОБРАБОТКИ ФАЙЛОВ В MS DOS

2.1. Цель работы:

Изучить методы синхронизации и взаимодействия процессов в мультипрограммных ОС на примере средств, предоставляемых MS DOS для блокировки одновременного доступа к файлам.

2.2. Введение

Сетевые (Netware) и мультипрограммные операционные системы (Windows, Unix и др.) обеспечивают возможность параллельной обработки данных. В данной лабораторной работе предлагается изучить возможности совместной обработки файлов на базе средств, предоставляемых ОС Windows в режиме эмуляции ОС MS DOS.

2.3. Методика выполнения работы

1) Изучить методические указания к работе.
2) Создать в среде Borland Pascal (Borland C) две программы, обеспечивающие совместное формирование файла по правилам, заданным в варианте задания.

Обе программы запускаются одновременно, первая программа создает пустой) файл, первая (вторая) программа формирует очередную (очередные) запись (записи) файла, вторая (первая) программа считывает эту (эти) запись (записи) и, в свою очередь, формирует следующую (следующие) запись (записи).

Запись файла имеет следующую структуру:

Номер записи	Номер программы	Данные

В задании определяется
- последовательность работы программ;

- количество записей, создаваемых каждой программой;
- правило формирования очередной записи.

2.4. Варианты заданий

Варианты заданий представлены в таблице 2.1.

Таблица 2.1 – Варианты заданий к лабораторной работе №2

№ вар.	Очередность работы программ	Количество формируемых записей	Обработка данных
1	1-2	1-1	Сумма 2-х пред. записей
2	1-2	2-1	- « -
3	1-2	1-2	- « -
4	1-2	2-2	- « -
5	1-2	1-1	Ср. арифм. 2-х пред. записей
6	1-2	2-1	- « -
7	1-2	1-2	- « -
8	1-2	2-2	- « -
9	2-1	1-1	Сумма 2-х пред. записей
10	2-1	2-1	- « -
11	2-1	1-2	- « -
12	2-1	2-2	- « -
13	2-1	1-1	Ср. арифм. 2-х пред. записей
14	2-1	2-1	- « -
15	2-1	1-2	- « -
16	2-1	2-2	- « -
17	1-2	1-1	Сумма всех пред. записей
18	1-2	2-1	- « -
19	1-2	1-2	- « -
20	1-2	2-2	- « -
21	2-1	1-1	Ср. ариф. всех пред. записей
22	2-1	2-1	- « -
23	2-1	1-2	- « -
24	2-1	2-2	- « -

2.5. Содержание отчета

Постановка задачи. Описание программы. Методика и результаты испытаний. Выводы.

2.6. Контрольные вопросы

1. Понятие файла. Сформулируйте понятия дескриптор файла, файловая переменная, запись, смещение.
2. Совместная обработка файлов. Режимы открытия файла.
3. Функции MS DOS. Правила работы с функцией 5С.
4. Понятие критического ресурса и критической области.
5. Постановка задачи «производитель-потребитель» исключения.

2.7. Рекомендации по выполнению работы

2.7.1. Совместная обработка файлов

Операционная система MS DOS относится к однопрограммным операционным системам. Как следствие, в обычном режиме работы одновременный доступ к одному и тому же файлу невозможен (так как невозможен одновременный запуск двух и более программ). Однако обрабатываемый файл может находиться не на локальной машине, а в сети – на файловом сервере. В этом случае файл может обрабатываться одновременно несколькими программами.

В нашей задаче каждая из программ должна обеспечивать строгую очередность при работе с файлом: сформировать свою запись, затем дождаться, пока другая программа не сформирует свою запись и только после этого формировать очередную запись. Другими словами, программы осуществляют последовательный двусторонний обмен данными, используя общий файл.

2.7.2. Постановка задачи «производитель-потребитель».

В простейшем случае взаимодействуют два процесса с жестко распределенными между ними функциями: первый процесс формирует сообщение, предназначенное для второго процесса, и помещает его в общую область данных (по смыслу являющуюся критическим ресурсом). Второй процесс (потребитель) принимает это сообщение, освобождая общую область для посылки очередного сообщения первым процессом (производителем) [1].

Необходимо согласовать работу двух параллельных процессов при одностороннем (в простейшем случае) обмене сообщениями по мере

развития процессов таким образом, чтобы удовлетворить следующим требованиям:

– выполнять требования задачи взаимного исключения по отношению к общей области данных;
– учитывать состояние общей области данных, характеризующее возможность или невозможность посылки (принятия) очередного сообщения.

2.7.3. Фрагмент программы для записи данных в файл

```
i:=0;
filemode:=$02;  { задать монопольный доступ к файлу}
while i<100 do begin { формируем по 100 записей}
{$I-}
 repeat
    reset(f);
    { открываем файл – захватываем его}
 until ioresult<>0;
{$I+}
if filesize(f) mod 2 = 0 then
begin
     { например, если количество записей четное }
     write(f,d);
     inc(i);
     { формируем очередную запись }
end;
close(f);
{закрываем файл – освобождаем его }
end;
```

3. ЛАБОРАТОРНАЯ РАБОТА №3. СИНХРОНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

3.1. Цель работы:

Изучить методы синхронизации и взаимодействия процессов в мультипрограммных ОС на примере многопоточных приложений среды Delphi и C++ Builder.

3.2. Введение

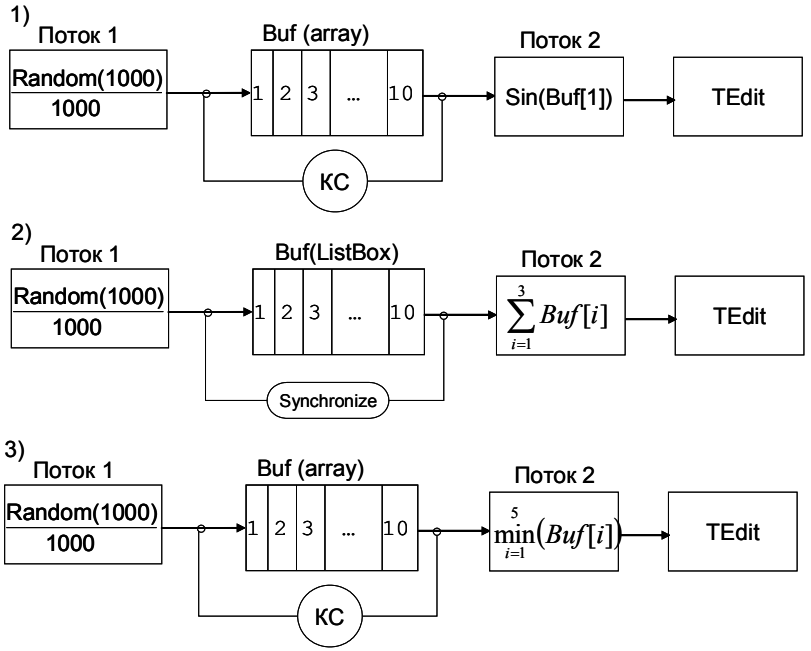
Мультипрограммные операционные системы (Windows, Unix и др.) предоставляют приложению некоторый интервал времени центрального процессора и в момент, когда приложение переходит к ожиданию

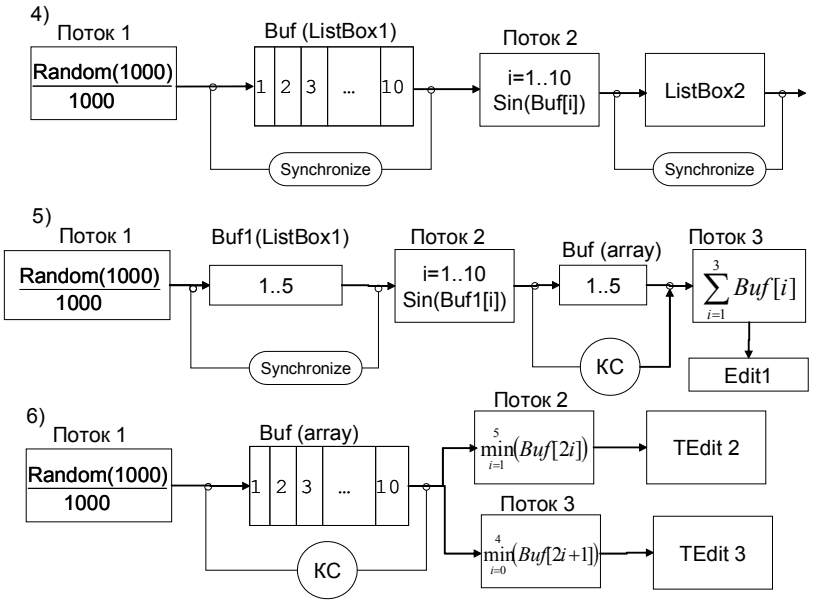
сообщений или освобождает процессор, операционная система передает управление другой задаче. В данной лабораторной работе предлагается на базе средств управления потоками среды Delphi (C++ Builder) изучить механизмы синхронизации и управления процессами.

3.3. Методика выполнения работы

1) Изучить методические указания к работе.
2) Создать в среде Delphi (C++ Buider) многопоточное приложение, описанное в разделе «Пример создания многопоточного приложения в Delphi».
3) Изменить текст приложения в соответствии с вариантом задания к лабораторной работе.

3.4. Варианты заданий





3.5. Содержание отчета

Постановка задачи. Описание программы. Методика и результаты испытаний. Выводы.

3.6. Контрольные вопросы

1. Понятие процесса. Сформулируйте понятия дескриптор процесса, контекст процесса.
2. Понятие потока. Отличие потока от процесса.
3. Гонки и тупики. Методы синхронизации потоков.
4. Приоритеты потоков, классы потоков.
5. Средства управления потоками в среде визуального программирования Delphi (C++ Builder).

3.7. Рекомендации по выполнению работы

3.7.1. Понятие потока

Потоки — это объекты, получающие время процессора. Время

процессора выделяется квантами (около 19 мс). Квант времени — это интервал, имеющийся в распоряжении потока до тех пор, пока время не будет передано в распоряжение другого потока. Кванты выделяются не программам или процессам, а порожденным ими потокам. Как минимум, каждый процесс имеет хотя бы один (главный) поток, но операционные системы Windows 95 и Windows NT позволяют запустить в рамках процесса сколько угодно потоков.

Существуют два главных типа потоков — асимметричные и симметричные. Асимметричные потоки (asymmetric threads) — это потоки, решающие различные задачи и, как правило, не разделяющие совместные ресурсы. Один из таких потоков отвечает за печать; другой обрабатывает сообщения от клавиатуры и мыши; третий заведует автоматическим сохранением документа пользователя.Симметричные потоки (symmetric threads) выполняют одну и ту же работу, разделяют одни ресурсы и исполняют один код [2].

3.7.2. Типичные ошибки при использовании потоков

Как и при использовании других сильнодействующих средств, в отношении потоков вы должны соблюдать определенные правила безопасности. Две типичные проблемы, с которыми программист может столкнуться при работе с потоками, — это гонки (race conditions) и тупики (deadlocks).

3.7.3. Гонки

Ситуация гонок возникает, когда два или более потока пытаются получить доступ к общему ресурсу и изменить его состояние. Рассмотрим следующий пример. Пусть Поток 1 формирует случайные числа и помещает их в буфер из bufsize элементов по следующему алгоритму:

```
if BufCount<BufSize then
  begin
    BufCount:=BufCount+1; //Увеличение числа элементов
    Buf[BufCount]:=Random(100);
  end;
```

Поток 2 извлекает числа из буфера и находит их сумму:

```
if BufCount>0 then
  begin
```

```
    sum:=sum+Buf[1];
    for j:=1 to BufCount-1 do  //Сдвиг буфера на 1 элемент влево
            Buf[j]:=Buf[j+1];
    BufCount:=BufCount-1; //Уменьшение числа элементов
  end;
```

Пусть число элементов буфера равно 5. Поток 2 извлекает 1-й элемент, добавляет его к сумме и выполняет сдвиг оставшихся четырех элементов. После этого квант времени, выделенного потоку заканчивается. Реальное число элементов буфера – 4, пятая ячейка свободна. Значение переменной BufCount потоком 2 не изменено, в связи с этим при передаче управления потоку 1 новое значение будет записано не в 5-ю ячейку, а 6-ю, что приведет к ошибке при вычислении суммы. Такая ситуация называется гонкой и для ее решения используются средства синхронизации потоков: критические секции и семафоры.

3.7.4. Тупики

Тупики имеют место, когда поток ожидает ресурс, который в данный момент принадлежит другому потоку. Рассмотрим пример: Поток 1 захватывает объект А и, для того, чтобы продолжать работу, ждет возможности захватить объект Б. В то же время Поток 2 захватывает Объект Б и ждет возможности захватить Объект А. Развитие этого сценария заблокирует оба потока; ни один из них не будет исполняться. Возникновения как ситуаций гонок, так и тупиков можно избежать, если использовать средства синхронизации потоков.

3.7.5. Класс TThread

Delphi представляет программисту полный доступ к возможностям программирования интерфейса Win32. Класс TThread предоставляет разработчику доступ к программированию потоков, обеспечивая гарантию совместимости с библиотекой визуальных компонентов VCL. Без использования класса TThread во время вызовов VCL могут возникнуть ситуации гонок.

Нужно отдавать себе отчет, что с точки зрения операционной системы, поток — это ее объект. При создании он получает дескриптор и отслеживается ОС. Объект класса TThread — это конструкция Delphi, соответствующая потоку ОС. Этот объект VCL создается до реального возникновения потока системе и уничтожается после его исчезновения.

Изучение класса TThread начнем с конструктора:

constructor Create(CreateSuspended: Boolean);

В качестве аргумента он получает параметр CreateSuspended. Если его значение равно True, вновь созданный поток не начинает выполняться до тех пор, пока не будет сделан вызов метода Resume. В случае, если CreateSuspended имеет значение False, поток начинает исполнение и конструктор завершается.

destructor Destroy; override;

Деструктор Destroy вызывается, когда необходимость в созданном потоке отпадает. Деструктор завершает его и высвобождает все ресурсы, связанные с объектом TThread.

procedure Resume;

Метод Resume класса TThread вызывается, когда поток возобновляется после остановки, или если он был создан с параметром createsuspended, равным True.

procedure Suspend;

Вызов метода suspend приостанавливает поток с возможностью запуска впоследствии. Метод suspend приостанавливает вне зависимости от кода, исполняемого потоком в данный момент; выполнение продолжается с точки останова.

property Suspended: Boolean;

Свойство suspended позволяет программисту определить, не приостановлен ли поток. С помощью этого свойства можно также запускать и останавливать поток. Установив suspended в True, вы получите тот же результат, что и при вызове метода suspend — приостановку. Наоборот, установка suspended в False возобновляет выполнение потока, как и вызов метода Resume.

function Terminate: Integer;

Для окончательного завершения потока (без последующего запуска) существует метод Terminate; он останавливает поток и возвращает управление вызвавшему процессу только после того, как это произошло. Значение, возвращаемое функцией Terminate, соответствует состоянию потока. Примерами возможных состояний являются случай нормального завершения и случай, когда к моменту вызова Terminate поток уже завершился (или был завершен из другого потока). Метод Terminate автоматически вызывается и из деструктора объекта. В явном виде его,

за редким исключением, вызывать не надо.

Property Terminated: Boolean;

Свойство Terminated позволяет узнать, произошел ли уже вызов метода Terminate или нет.

Function WaitFor: Integer;

Метод WaitFor предназначен для синхронизации и позволяет одному потоку дождаться момента, когда завершится другой поток. Если вы внутри потока FirstThread пишете код: Code := SecondThread.WaitFor. Это это означает, что поток FirstThread останавливается до момента завершения потока SecondThread. Метод WaitFor возвращает код завершения ожидаемого потока.

property Handle: THandle read FHandle;

property ThreadID: THandle read FThreadID;

Свойства Handle и ThreadiD дают программисту непосредственный доступ к потоку средствами API Win 32. Если разработчик хочет обратиться к потоку и управлять им, минуя возможности класса TThread, значения Handle и ThreadID могут быть использованы в качестве аргументов функций Win 32 API. Например, если программист хочет перед продолжением выполнения приложения дождаться завершения сразу нескольких потоков, он должен вызвать функцию API waitForMuitipieObjects; для ее вызова необходим массив дескрипторов потоков.

property Priority: TThreadPriority;

Свойство priority позволяет запросить и установить приоритет потоков. Приоритет определяет, насколько часто поток получает время процессора. Естественно, программист захочет выделить главному потоку в приложении большее время, а потоку, например, с фоновой проверкой орфографии — меньшее. Допустимыми значениями приоритета являются tpidle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest и tpTimeCritical. Будьте осторожны, используя приоритеты tpHighest и tpTimeCritical. Оба они могут оказать влияние на выполнение приложения, а последний — и на всю операционную систему.

procedure Synchronize(Method: TThreadMethod);

Этот метод относится к секции protected, то есть может быть вызван

только из потомков TThread. Delphi предоставляет программисту метод synchronize для безопасного вызова методов VCL внутри потоков. Во избежание ситуаций гонок, метод Synchronize дает гарантию, что к каждому объекту VCL одновременно имеет доступ только один поток. Аргумент, передаваемый в метод synchronize, — это имя метода, который производит обращение к VCL; вызов synchronize с этим параметром — это то же, что и вызов самого метода. Такой метод (класса TThreadMethod) не должен иметь никаких параметров и не должен возвращать никаких значений. К примеру, в основной форме приложения нужно предусмотреть функцию

```
procedure TMainFom.SyncShowMessage;
begin
ShowMessage(IntToStr(Listi.Count)) ;
//другие обращения к VCL
end;

а в потоке для показа сообщения писать не
ShowMessage(IntToStr(List1.Count)) ;
А только так:
Synchronize(MainFonn.SyncShowMessage);
```

Производя любое обращение к объекту VCL из потока, убедитесь, что при этом используется метод Synchronize; в противном случае результаты могут оказаться непредсказуемыми.

Procedure Execute; virtual; abstract;

Это и есть главный метод объекта TThread. В его теле должен содержаться код, который и представляет собой собственно поток. Метод Execute класса tThread объявлен как абстрактный.

Переопределяя метод Execute, мы можем тем самым закладывать в новый потоковый класс то, что будет выполняться при его запуске. Если поток был создан с аргументом CreateSuspended, равным False, то метод Execute выполняется немедленно, в противном случае Execute выполняется после вызова метода Resume.

Если поток рассчитан на однократное выполнение каких-либо действий, то никакого специального кода завершения для него писать не надо. После выполнения метода Execute будет вызван деструктор, который сделает все необходимое.

Если же в потоке будет выполняться какой-то цикл, и поток должен

завершиться вместе с приложением, то условия окончания цикла должны быть примерно такими:

```
procedure TMyThread.Execute;
begin
repeat
DoSomething;
Until CancelCondition or Terminated;
End;
```

здесь CancelCondition — ваше личное условие завершения потока (исчерпание данных, поступление на вход того или иного символа и т. п.), свойство Terminated говорит о завершении потока извне (скорее всего, завершается породивший его процесс).

С завершением потока надо быть очень внимательным, — если он зациклился и не реагирует на сигналы завершения, то зависнет все приложение.

property ReturnValue: Integer;

Свойство ReturnValue позволяет узнать и установить значение, возвращаемое потоком по его завершении. Эта величина полностью определяется пользователем. По умолчанию поток возвращает ноль, но если программист захочет вернуть другую величину, то простая переустановка свойства ReturnValue внутри потока позволит получить эту информацию другим потокам.

3.7.6. Средства синхронизации потоков

Проще всего говорить о синхронизации, если создаваемый поток не взаимодействует с ресурсами других потоков и не обращается к VCL. Допустим, у вас на компьютере несколько процессоров, и вы хотите распараллелить вычисления. Тогда вполне уместен следующий код:

```
MyCompThread := TComputationThread.Create( False );
//пока поток считает, можно выполнить часть работы
DoSomeWork;
//а теперь ждем завершения потока
MyCompThread.WaitFor;
```


Приведенная схема совершенно недопустима, если во время своей работы поток MyCompThread обращается к VCL посредством метода synchronize. В этом случае поток ждет главный поток для обращения к VCL, а тот, в свою очередь, его, что приводит к ситуации, которая называется «тупик».

Чтобы избежать тупика следует использовать программный интерфейс Win32. Он предоставляет богатый набор инструментов, которые могут понадобиться для организации совместной работы потоков.

Главные понятия для понимания механизмов синхронизации — функции ожидания и объекты ожидания. В API предусмотрен ряд функций, позволяющих приостановить выполнение вызвавшего эту функцию потока вплоть до того момента, как будет изменено состояние какого-то объекта, называемого объектом ожидания. (Под этим термином здесь понимается не объект Delphi, а объект операционной системы.) Простейшая из этих функций WaitForSingleObject — предназначена для ожидания одного объекта.

К возможным вариантам относятся четыре объекта, разработанных специально для синхронизации: событие (event), взаимное исключение (mutex), семафор (semaphore) и таймер (timer). К ним может быть добавлена критическая секция (critical section). Кроме них, можно ждать и других объектов, дескриптор которых используется в основном для других целей, но может применяться и для ожидания. К ним относятся: процесс, поток, оповещение об изменении в файловой системе (change notification) и консольный ввод.

Перечисленные выше средства синхронизации в основном инкапсулированы в состав классов Delphi. У программиста есть две альтернативы. С одной стороны, в состав библиотеки VCL включен модуль SYNCOBJS.PAS, содержащий классы для события (TEvent) и критической секции (TCriticalSection). С другой, с Delphi поставляется отличный пример IPCDEMOS, который иллюстрирует проблемы взаимодействия процессов и содержит модуль IPCTHRD.PAS с аналогичными классами — для того же события, взаимного исключения (TMutex), а также совместно используемой памяти (TSharedMem).

3.7.7. Взаимные исключения

Объект типа взаимное исключение (Mutex) позволяет только одному потоку в данное время владеть им. Если продолжать аналогии, то этот объект можно сравнить с эстафетной палочкой.

Класс, инкапсулирующий взаимное исключение — TMutex находится в модуле IPCTHRD.PAS (пример IPCDEMOS). Конструктор:

constructor Create(const Name: string);

задает имя создаваемого объекта. Первоначально он не принадлежит никому. (Но функция API createMutex, вызываемая в нем, позволяет передать созданный объект тому потоку, в котором это произошло.) Далее метод:

function Get(TimeOut: Integer): Boolean;

производит попытку в течение Timeout миллисекунд завладеть объектом (в этом случае результат равен True). Если объект более не нужен, следует вызвать метод:

function Release: Boolean;

Программист может использовать взаимное исключение, чтобы избежать считывания и записи общей памяти несколькими потоками одновременно.

3.7.8. Критическая секция

Работая в Delphi, программист может также использовать объект типа критическая секция (critical section). Критические секции подобны взаимным исключениям по сути, однако между ними существуют два главных отличия:

- взаимные исключения могут быть совместно использованы потоками в различных процессах;
- если критическая секция принадлежит другому потоку, ожидающий поток блокируется вплоть до освобождения критической секции. В отличие от этого, взаимное исключение разрешает продолжение по истечении тайм-аута.

Критические секции и взаимные исключения очень схожи. На первый взгляд, выигрыш от использования критической секции вместо взаимного исключения не очевиден. Критические секции, однако, более эффективны, чем взаимные исключения, так как используют меньше системных ресурсов. Взаимные исключения могут быть установлены на определенный интервал времени, по истечении которого выполнение продолжается; критическая секция всегда ждет столько, сколько потребуется.

Возьмем класс TCriticaisection (модуль SYNCOBJS.PAS). Логика использования его проста — "держать и не пускать". В многопотоковом приложении создается и инициализируется общая для всех потоков критическая секция. Когда один из потоков достигает критически

важного участка кода он пытается захватить секцию вызовом метода Enter:

```
MySection.Enter;
try
    DoSomethingCritical ;
finally
    MySection.Leave;
end;
```

Когда другие потоки доходят до оператора захвата секции и обнаруживают, что она уже захвачена, они приостанавливаются вплоть до освобождения секции первым потоком путем вызова метода Leave. Обратите внимание, что вызов Leave помещен в конструкцию try. .finally — здесь требуется повышенная надежность. Критические секции являются системными объектами и подлежат обязательному освобождению — впрочем, как и остальные рассматриваемые здесь объекты.

3.7.9. Пример создания многопоточного приложения в Delphi

Рассмотрим пример программы, которая включает в себя два потока. Первый генерирует случайные числа и помещает их в буфер из BufSize элементов, а второй – находит их сумму (рисунок 3.1). Результат выводится в окно редактора TEdit.

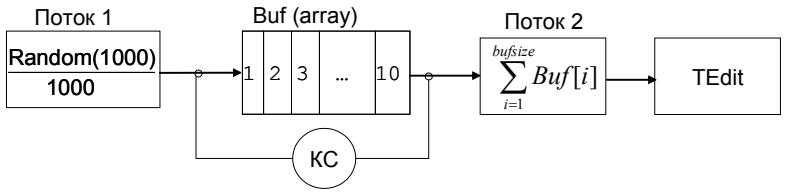


Рисунок 3.1 – Схема вычислительного процесса

Экранная форма для приложения представлена на рисунке 3.2.

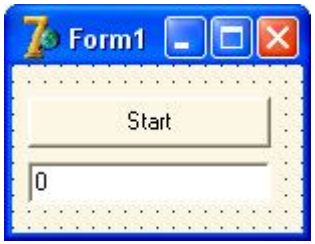


Рисунок 3.2 – Экранная форма приложения

Определим типы данных для потоков Thr1 и Thr2:

```
TThr1 = class(TThread)
private
  { Private declarations }
protected
  procedure Execute; override;
end;

TThr2 = class(TThread)
private
  { Private declarations }
protected
  procedure Execute; override;
end;
```

и следующие глобальные переменные:

```
Thr1 : TThr1; // первый поток
Thr2 : TThr2; // второй поток
CS   : TCriticalSection; //критическая секция
Buf  : array[1..BufSize] of integer;  //буфер
BufCount : integer; //количество элементов буфера
Sum  : Integer; //сумма
```

Первый поток генерирует случайные числа и помещает их в буфер. При этом необходимо следить за тем, чтобы буфер не переполнялся и защититься от гонок.

```
procedure TThr1.Execute;
```



```
var i : integer;
begin
 i:=0;
 while i<ExpCount do  //Генерируем 1000 чисел
  begin
   if BufCount<BufSize then //Проверяем заполненность буфера
     begin
       Cs.Enter;  //Вход в кририческую секцию
       BufCount:=BufCount+1; //Увеличиваем число элементов
       Buf[BufCount]:=Random(1000); // Генерируем элемент
       inc(i);
       Cs.Leave;  //Выходим из критической секции
     end;
  end;
end;
```

Второй поток извлекает числа из буфера (если они там есть) и находит их сумму. При завершении вычислений результат выводится в TEdit.

```
procedure TThr2.Execute;
var i,j : integer;
begin
 i:=0;
 while i<ExpCount do //Суммируем ExpCount чисел
  begin
   if BufCount>0 then  //Если в буфере что-то есть
     begin
       Cs.Enter; //Входим в критическую область
       sum:=sum+Buf[1]; //Суммируем
       for j:=1 to BufCount-1 do   Buf[j]:=Buf[j+1]; //Сдвиг буфера
       BufCount:=BufCount-1; //Уменьшаем число элементов на 1
       inc(i);
       Cs.Leave; //Выходим из критической области
     end;
  end;
  Synchronize(Form1.Showres);  //Вывод результата – используем
                               //метод  Synchronize
end;
```

Реакция на кнопку «Старт»:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text:='0'; Thr1:=TThr1.Create(True); //Создаем потоки
  Thr2:=TThr2.Create(True);
  Thr1.FreeOnTerminate:=True;  Thr2.FreeOnTerminate:=True;
  BufCount:=0;  Sum:=0; //Очищаем буфер
  Thr1.Resume;  Thr2.Resume; //Запускаем потоки на выполнение
end;
```

Создание критической секции можно произвести в обработчике события FormCreate:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 CS:=TCriticalSection.Create;
end;
```

4. ЛАБОРАТОРНАЯ РАБОТА №4. СИНХРОНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ С ИСПОЛЬЗОВАНИЕМ СЕМАФОРОВ

4.1. Цель работы:

Изучить методы синхронизации параллельных процессов с использованием семафоров.

4.2. Введение

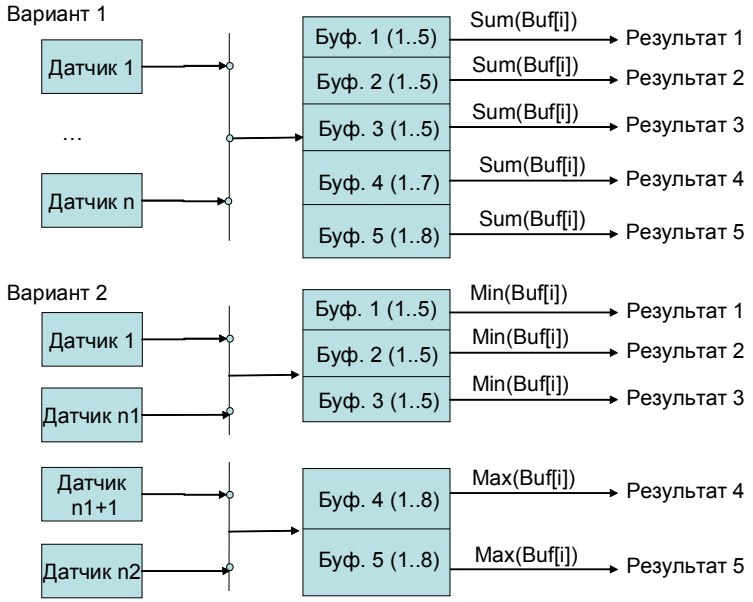
Semaphore (семафор) - напоминает критическую секцию, но разрешает захват ресурса не одним, а несколькими потоками, причем число потоков ограничено доступным количеством ресурса. При создании семафора задается количество единиц разделяемого ресурса (aMaxCount). При создании можно также захватить одну или больше единиц ресурса (aInitialCount), если аргумент равен aMaxCount, то все ресурсы свободны. Для захвата ресурса процесс вызывает метод Wait, для освобождения ресурса - метод Release. За один раз можно освободить одну или несколько единиц ресурса, но захватить можно только одну за один раз. Пример использования семафора - торговый прилавок с тремя продавцами. Одновременно может быть обслужено не более 3х клиентов: остальные клиенты должны ждать освобождения ресурса -

продавца.

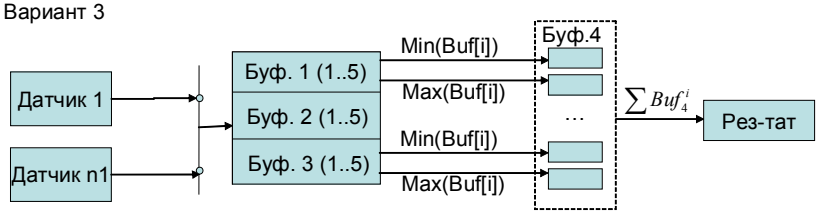
4.3. Методика выполнения работы

1) Изучить методические указания к работе.
2) Разработать программу, моделирующую параллельный прием информации от нескольких датчиков в многоканальную память и ее последующую обработку в соответствии с вариантом задания.
3) Исследовать эффективность использования буферов в зависимости от числа потоков приема/обработки.

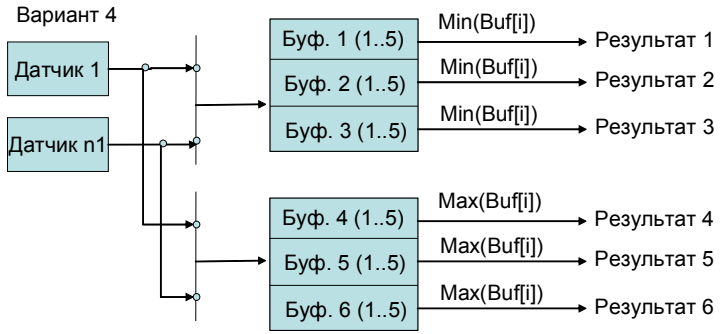
4.4. Варианты заданий



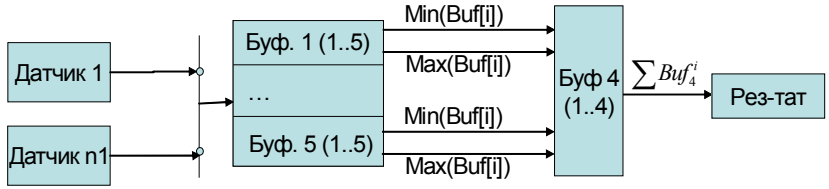
Вариант 3



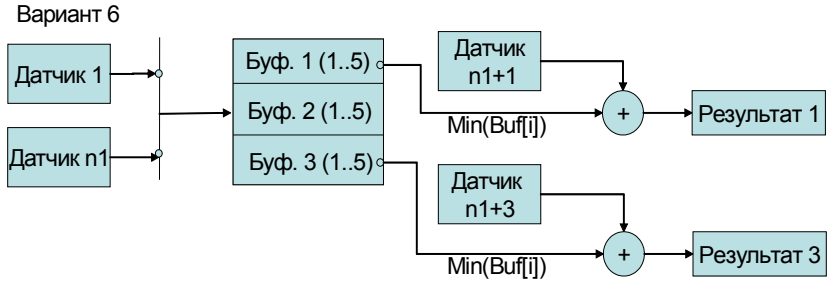
Вариант 4



Вариант 5



Вариант 6



4.5. Содержание отчета

Постановка задачи. Описание программы. Методика и результаты испытаний. Выводы.

4.6. Контрольные вопросы

1. Перечислите общее и различия в функционировании семафоров и критических секций
2. Каким образом с помощью семафоров можно решить классическую задачу «читатели и писатели»?
3. Перечислите функции Win32 для управления семаформи.
4. В чем различие в функциях WaitForSingleObject и WaitForMultipleObject?

4.7. Рекомендации по выполнению работы

При разработке приложений, выполняющих обработку информации, поступающую от различных датчиков часто возникает задача оптимизации затрат процессорного времени и памяти. При сравнительно небольшой интенсивности событий приема информации и достаточно больших ее объемах целесообразно использовать многоканальную память, назначая активному датчику свободный канал.

Блок-диаграмма такой системы представлена на рисунке 4.1.

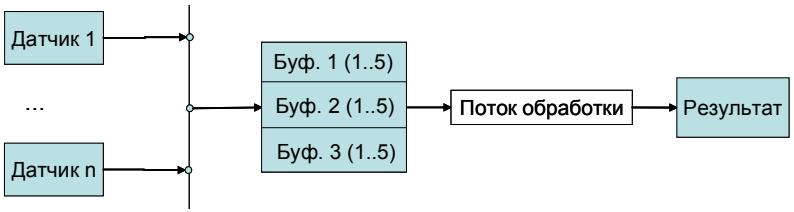


Рисунок 4.1 – Блок-схема системы с многоканальной памятью

Логика работы такой системы описывается следующим алгоритмом. Перед приемом информации от датчика менеджер многоканальной памяти выделяет датчику свободный на данный момент диапазон адресов и контролирует прием данных. Поток обработки последовательно просматривает все буфера и выполняет соответствующую обработку находящейся там информации. При этом требуется обеспечить соблюдение следующих ограничений:

1) Количество датчиков, одновременно работающих с многоканальной памятью не должно превышать числа каналов (буферов).
2) Недопустимо единовременное назначение двум датчикам адресов из одного и того же или пересекающихся диапазонов.
3) Поток обработки может выполнять вычисления только над данными, которые уже окончательно сформированы.
4) Буфер (диапазон адресов) становится свободным только при завершении процесса обработки всех данных из буфера.

Для ограничения числа потоков к разделяемому ресурсу удобно использовать механизм синхронизации, называемый семафором. Формальное определение семафора предложил Дейкстра, который ввел два новых примитива. В абстрактной форме эти примитивы, обозначаемые P и V, оперируют над целыми неотрицательными

переменными, называемыми семафорами. Пусть S такой семафор. Операции определяются следующим образом:

V(S) : переменная S увеличивается на 1 одним неделимым действием; выборка, инкремент и запоминание не могут быть прерваны, и к S нет доступа другим процессам во время выполнения этой операции.

P(S) : уменьшение S на 1, если это возможно. Если S=0, то невозможно уменьшить S и остаться в области целых неотрицательных значений, в этом случае процесс, вызывающий P-операцию, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также является неделимой операцией. В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную. Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания, в то время как V-операция может при некоторых обстоятельствах активизировать другой процесс, приостановленный операцией P.

Таким образом, реализация ограничения 1 возможно с применением механизма семафоров, а ограничений 2-4 с использованием критической секции.

Платформа Win32 предоставляет программисту следующий инструментарий для работы с семафорами.

Создание семафора:

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

Здесь параметр LPSECURITY_ATTRIBUTES (атрибуты защиты) обычно принимает значение Nil, lMaximumCount – максимальное количество единиц ресурса, lInitialCount – начальное количество свободного ресурса, lpName – имя семафора в ОС (семафор может использоваться не только для синхронизации потоков, но и приложений).

Функция BOOL ReleaseSemaphore(

```
   HANDLE hSemaphore, // handle of the semaphore object
   LONG lReleaseCount,   // amount to add to current count
   LPLONG lpPreviousCount // address of previous count
```

) выполняет освобождение lReleaseCount единиц ресурса,

охраняемого семафором hSemaphore. При необходимости можно узнать предыдущее значение семафора, передавая в функцию указатель на переменную lpPreviousCount.

Ниже перечисленные функции позволяют занимать единицу ресурса одного или нескольких объектов семафора

```
DWORD WaitForSingleObject(
  HANDLE hHandle,  // handle of object to wait for
  DWORD dwMilliseconds   // time-out interval in milliseconds
 );
DWORD WaitForMultipleObjects(

  DWORD nCount,    // number of handles in the object handle array
  CONST HANDLE *lpHandles,    // pointer to the object-handle array
  BOOL bWaitAll,// wait flag
  DWORD dwMilliseconds   // time-out interval in milliseconds    );
```

Объект семафора считается доступным, если его внутреннее состояние больше 0.

Разработку программы начнем с создания интерфейса, представленного на рисунке 4.2.

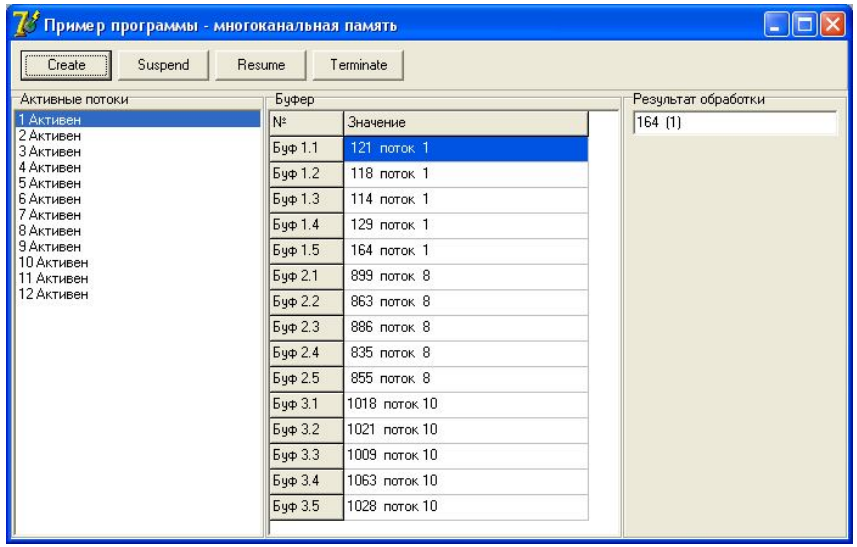


Рисунок 4.2 – Интерфейс для управления и исследования работы

параллельных потоков

На интерфейсе размещены:

1) Кнопки управления потоками-моделями датчиков: Create (Создание потока - подключение нового датчика), Suspend (приостановка - временное отключение датчика), Resume (запуск потока – прием данных от датчика) и Terminate (завершение – отключение датчика).
2) Компонент ListBox1:TListBox для визуализации и управления состоянием потоков.
3) Компонент SG1:TStringGrid для отображения текущего состояния ячеек многоканальной памяти
4) Edit1:TEdit для вывода результата обработки.

Объявим глобальные переменные:

Const BufFree : array [1..3] of boolean=(True,True,True); - массив признаков свободы каналов памяти.

```
var
  Form1: TForm1; //Интерфейс
  Sem  : integer; //Семафор
  CS   : TCriticalSection; //Критическая секция
  Buf        :    array[1..15]    of    integer;
//Многоканальная память 3x5 ячеек
  Id   : integer; //Идентификатор потока
  wt   : TWorkThrd; //Поток обработки
```

При создании интерфейсной формы не забудем создать все необходимые объекты:

```
  Sem:=CreateSemaphore(Nil,3,3,'MySem');
  CS :=TCriticalSection.Create;
  wt:=TWorkThrd.Create(False);
  Id := 1;
```

Теперь мы можем описать поведение датчика и потока обработки в терминах параллельных процессов:

```
//Класс потока – модели датчика
  TGenThrd = class(TThread)
  private
    GenMas : array [1..5] of integer; //Временный
массив данных датчика
    Chanel : integer;      //Номер канала памяти,
выделенного датчику
```

```
      procedure MoveToBuf;

    protected
      procedure Execute; override;
    public
      Id     : integer;
    end;


  //Класс потока  обработки
    TWorkThrd = class(TThread)
    private
      { Private declarations }
      s : string;
      procedure ShowResult;
    protected
      procedure Execute; override;
    end;


  procedure TGenThrd.Execute;
  var i : integer;
  begin
    while (not terminated) do
    begin
       //Ждем семафор
      while
(WaitForSingleObject(Sem,1000)<>WAIT_OBJECT_0)  do;
       //Имитируем данные от датчика
        for i:=1 to 5 do
  GenMas[i]:=Random(100)+id*100;
       //Занимаем критическую секцию
        CS.Enter;
      //Находим свободный канал памяти
        Chanel:=FindFreeBuf;
      //Переписываем в него данные
        for i:=1 to 5 do
            Buf[i+(Chanel-1)*5]:=GenMas[i];
    //Освобождаем критический ресурс
       CS.Leave;
  //Выводим результат
       Synchronize(MoveToBuf);
    end;
  end;
```



```
//Процедура обработки данных
procedure TWorkThrd.Execute;
var Chanel : integer;
    i,max  : integer;
begin
  while not terminated do
  begin
     CS.Enter;  //Заняли критический ресурс
     for Chanel:=1 to 3 do
        if not BufFree[Chanel] then
         //Нашли заполненный канал
          begin
             max:=Buf[(Chanel-1)*5+1];
             for i:=2 to 5 do
               if Buf[(Chanel-1)*5+i]>max then
               max:=Buf[(Chanel-1)*5+i];
                BufFree[Chanel]:=True;
                ReleaseSemaphore(Sem,1,Nil);
                s:=IntToStr(max)+'('
                +IntToStr(Chanel)+')';
          end;
     CS.Leave; //Освободили критический ресурс
     Synchronize(ShowResult); //Вывели результат
  end;
end;
```

5. ЛАБОРАТОРНАЯ РАБОТА №5. СИНХРОНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ С ИСПОЛЬЗОВАНИЕМ СОБЫТИЙ

5.1. Цель работы:

Изучить методы синхронизации параллельных процессов с использованием событий.

5.2. Введение

Объект типа событие — простейший выбор для задач синхронизации. Он подобен дверному звонку — звенит до тех пор, пока его кнопка находится в нажатом состоянии, извещая об этом факте

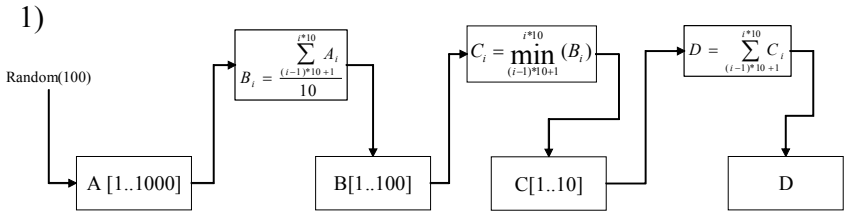
окружающих. Аналогично, и объект может находиться в двух состояниях, а "слышать" его могут многие потоки сразу. Событие может находиться в двух состояниях - активном (сигнализирущее состояние) и сброшенном (несигнализирующее состояние). В активное состояние событие переводится методом SetEvent, а сбрасывается методом ResetEvent. Событие может быть внутренним объектом потока, а может не принадлежать никакому потоку конкретно. Поток может ожидать только данного события с помощью метода Wait, либо использовать его дескриптор Handle для альтернативного ожидания.

5.3. Методика выполнения работы

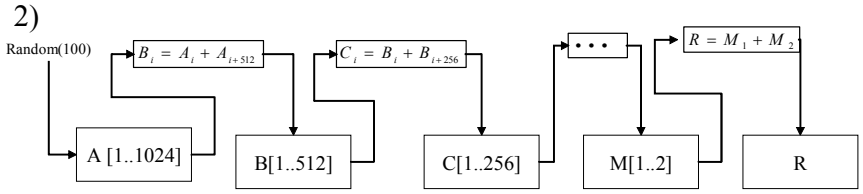
1. Изучить методические указания к лабораторной работе.
2. Разработать приложение, выполняющее конвейерную обработку исходного массива. Элементы конвейера синхронизируются событиями.

5.4. Варианты заданий

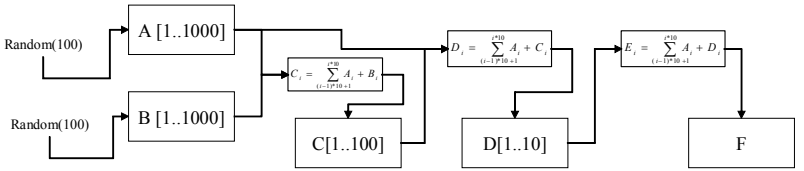
1)



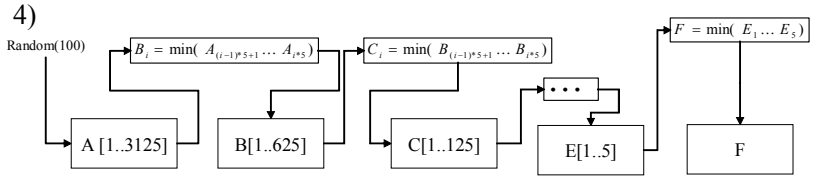
2)



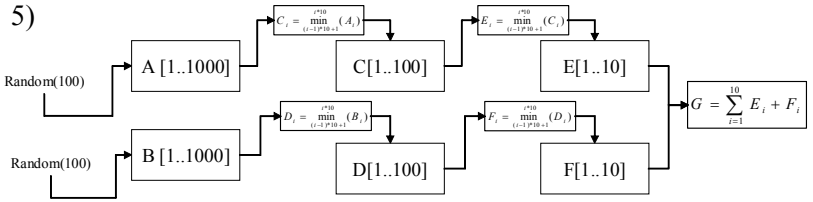
3)



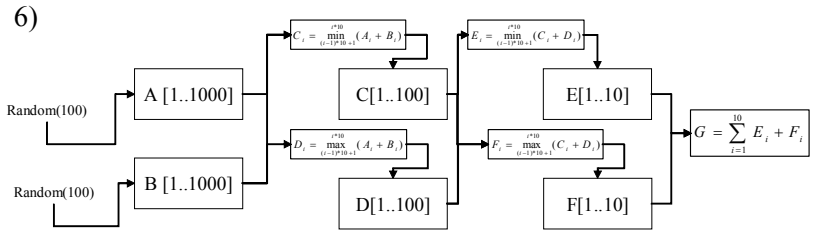
4)



5)



6)



5.5. Содержание отчета

Постановка задачи. Описание программы. Методика и результаты испытаний. Выводы.

5.6. Контрольные вопросы

1. Перечислите известные Вам средства синхронизации потоков?
2. В чем разница между событием и критической секцией?

3. Можно ли с помощью событий решить задачу о читателях и писателях без использования критической секции?
4. Как с помощью событий решить задачу «Обедающие философы»?

5.7. Рекомендации по выполнению работы

Класс TEvent (модуль SYNCOBJS.PAS) имеет два метода: setEvent и ResetEvent, переводящих объект в активное и пассивное состояние, соответственно. Конструктор имеет следующий вид:

constructor Create(EventAttributes: PSecurityAttributes; ManualReset, InitialState: Boolean; const Name: string);

Здесь параметр InitialState — начальное состояние объекта, ManualReset —способ его сброса (перевода в пассивное состояние). Если этот параметр равен True, событие должно быть сброшено вручную. В противном случае событие сбрасывается по мере того, как стартует хоть один поток, ждавший данный объект.

На третьем методе:

TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError) ;

Function WaitFor(Timeout: DWORD): TWaitResult;

остановимся подробнее. Он дает возможность ожидать активизации события в течение Timeout миллисекунд. Внутри этого метода происходит вызов функции WaitForSingleObject. типичных результатов на выходе waitFor два — wrSignaled, если произошла активизация события, и wrTimeout, если за время тайм-аута ничего не произошло. Если нужно ждать бесконечно долго, следует установить параметр Timeout в значение INFINITE.

Рассмотрим пример использования событий как средства синхронизации конвейерных вычислений. Пусть вычисление результата описывается следующей схемой (рисунок 5.1):

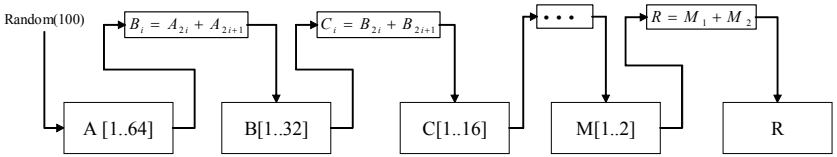


Рисунок 5.1 – схема конвейерной обработки данных

Заметим, что доступ к каждому буферу A, B, C … M, кроме R, имеют два потока: поток-писатель и поток-читатель. Потоки B,C и т.д. могут работать только в том случае, если в их распоряжении имеется как буфер-источник, так и буфер-приемник.

Вариант синхронизации работы 7 потоков и, соответственно, 7 буферов с использованием механизма событий представлен на рисунке 5.2.

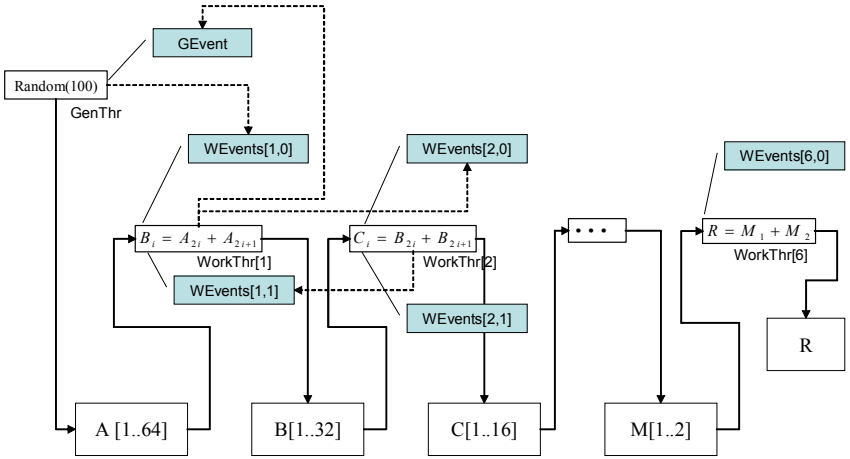


Рисунок 5.2 – схема синхронизации потоков

Событие GEvent сигнализирует потоку GenThr о том, что он может начать заполнение буфера A. После заверешения этой работы поток GenThr должен сигнализировать потоку WorkThr[1], что необходимые для его работы данные подготовлены. Поток WorkThr[1] может начать работу только тогда, когда все данные из буфера C будут обработаны потоком WorkThr[2]. Таким образом, обработка буфера A потоком WorkThr[1] будет осуществляться только в случае наличия двух сигналов: WEvents[1,0] (формируется потоком GenThr) и WEvents[1,1] (формируется потоком WorkThr[2]). Аналогичные рассуждения можно привести для потоков WorkThr[2].. WorkThr [5]. Для работы потока

WorkThr [6] достаточно наличия сигнала WEvents[6,0], однако для общности можно положить, что состояние сигнала WEvents[6,1] всегда равно True.

Таким образом, в системе существует два типа потоков: поток-генератор, синхронизируемый одним событием и поток-обработчик, синхронизируемый двумя событиями.

Потоку-генератору необходимо знать адрес буфера данных OutBuffer, его размер OutBufSize, указатель на событие разрешения работы RunEvent и указатель на событие StopEvent, которое поток-генератор должен установить при завершении своей работы.

```
TGenThr = class(TThread)
 private
   { Private declarations }
   procedure ShowBuf;
 protected
   procedure Execute; override;
 public
   OutBuffer  : TBuf;
   OutBufSize : integer;
   RunEvent   : TEvent;
   StopEvent  : TEvent;
 end;
```

Поток-обработчик должен знать адреса и размеры входного и выходного буферов данных, два указателя на событие разрешения работы RunEvent и два указателя на событие StopEvent, которые поток-обработчик должен установить при завершении своей работы .

```
TPackEvent=array[0..1] of TEvent;
TWorkThr = class(TThread)
private
  { Private declarations }
  procedure ShowBuf;
protected
  procedure Execute; override;
public
```



```
  InBuffer   : TBuf;
  OutBuffer  : TBuf;
  OutBufSize : byte;
  DemoRow    : integer;
  RunEvent   : TPackEvent;
  StopEvent  : TPackEvent;
end;
```

Действия потока-генератора:

```
procedure TGenThr.Execute;
var i : integer;
begin
 while (not terminated) do
 begin
   RunEvent.WaitFor(10000); //Ждем события-разрешения
   //Формируем буфер
   for i:=0 to OutBufSize-1 do
    OutBuffer[i]:={i;//}random(100);
   //Сброс события-разрешения
   RunEvent.ResetEvent;
   // разрешить работу следующему потоку
   StopEvent.SetEvent;     synchronize(ShowBuf);
 end;
end;
```

Действия потока-обработчика:

```
procedure TWorkThr.Execute;
var i : integer;
   wobj : array[0..1] of integer;
begin
 while (not terminated) do
 begin
   for i:=0 to 1 do    wobj[i]:=RunEvent[i].Handle;
   WaitForMultipleObjects(2,@wobj[0],true,10000);
   for i:=0 to OutBufSize-1 do
    OutBuffer[i]:=InBuffer[2*i]+InBuffer[2*i+1];
  //Формируем выходной буфер
   for i:=0 to 1 do
   begin
    //Сброс своих разрешений
```

```
        RunEvent[i].ResetEvent;
          if StopEvent[i]<>Nil then StopEvent[i].SetEvent;
        //Разрешить работу конкурентов
       end;
      synchronize(ShowBuf);
    end;
  end;
```

В вычислительной схеме будет один поток-генератор, одно событие разрешения гнерерации, 7 буферов, 6 потоков-обработчиков и 6 пар событий для их синхронизации.

```
  buf      : array of TBuf; // 7 буферов размером 64, 32, 16, 8, 4, 2, 1
  GenThr   : TGenThr;  //поток-генератор
  GEvent   : TEvent; //событие разрешения генерации
  WorkThr  : array [1..6] of TWorkThr; //6 потоков-обработчиков
  WEvents    : array [1..6] of TPackEvent; //события разрешения обработки
```

Далее представлен текст подпрограммы, выполняющей инициализацию переменных и расстановку синхронизационных переменных по потокам.

```
procedure TMainForm.Button1Click(Sender: TObject);
var i,j  : integer;
   cs   : integer;
begin
  GenThr:=TGenThr.Create(True);
  GEvent:=TEvent.Create(Nil,True,True,'GenEvent');
  GenThr.FreeOnTerminate:=true; cs:=64;
  setlength(buf,7);
  setlength(buf[0],cs);
  GenThr.OutBuffer:=Buf[0];
  GenThr.OutBufSize:=cs;
  GenThr.RunEvent:=GEvent;
  for i:=1 to 6 do
    begin
      WorkThr[i]:=TWorkThr.Create(True);
      WorkThr[i].FreeOnTerminate:=True;
```



```
      WorkThr[i].DemoRow:=i+1;
      for j:=0 to 1 do
     WEvents[i,j]:=TEvent.Create(Nil,True,(j mod 2)<>0,
                                          'WorkEvent'+IntToStr(i));
   end;
  GenThr.StopEvent:=WEvents[1,0];
  for i:=1 to 6 do
   begin
     cs:=cs div 2;
     SetLength(buf[i],cs);
     WorkThr[i].InBuffer:=Buf[i-1];
     WorkThr[i].OutBuffer:=Buf[i];
     WorkThr[i].OutBufSize:=cs;
     WorkThr[i].RunEvent[0]:=WEvents[i,0];
     WorkThr[i].RunEvent[1]:=WEvents[i,1];
     if i<6 then
       WorkThr[i].StopEvent[1]:=WEvents[i+1,0]
     else
       WorkThr[i].StopEvent[1]:=Nil;
     if i=1 then
        WorkThr[i].StopEvent[0]:=GEvent
      else
        WorkThr[i].StopEvent[0]:=WEvents[i-1,1];
   end;
   GenThr.Resume;
   for i:=1 to 6 do
     WorkThr[i].Resume;
end;
```

6. БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Богачев К.Ю. Основы параллельного программирования / К.Ю. Богачев – М.:Бином, 2001. – 336 с.

2. Сухарев М.В. Основы Delphi. Профессиональный подход / М.В. Сухарев. – М.:Наука и техника, 2004. – 623 с.

Заказ № от Тираж экз.

Изд-во СевНТУ