Министерство образования и науки Российской Федерации

Новосибирский государственный технический университет

ПРОЕКТИРОВАНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Методические указания к выполнению лабораторных работ для магистрантов направления 02.04.03 «Математическое обеспечение и администрирование информационных систем» факультета прикладной математики и информатики

Новосибирск

2021

СОДЕРЖАНИЕ

Лабораторная работа 1. Работа с последовательным портом персонального компьютера

1. Цель работы

Целью работы является изучение принципов работы и методов программирования COM портов и изучение различных алгоритмов вычисления контрольной суммы при передаче данных через COM порт

2. Методические указания

2.1 Принципы последовательной связи

Внешние устройства подключаются к ЭВМ через интерфейсы, называемые портами. Современные ЭВМ могут использовать следующие типы портов:

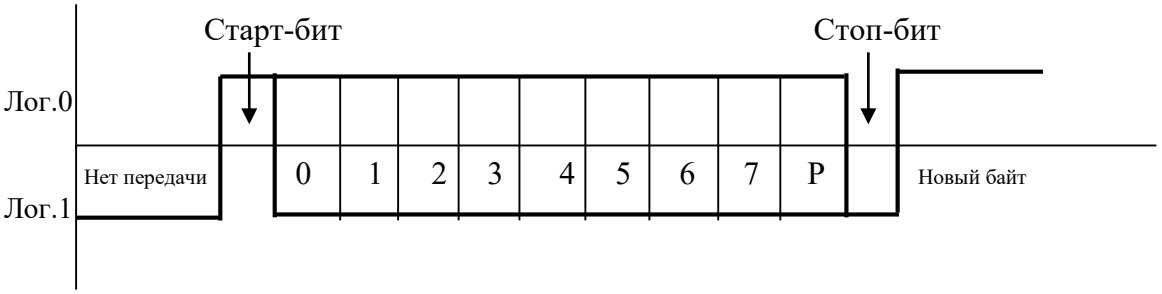
- параллельный порт LPT (разъем Centronics);
- последовательный порт COM (разъемы DB9 или DB25);
- последовательный порт USB (до 60 Мбит/с для USB 2.0);
- последовательный порт IEEE1394 (FireWare, iLink- для подключения цифровых видеокамер и других мультимедийных устройств, скорость – до 50 Мбит/с);
- последовательный порт PS/2;
- игровой порт MIDI (Gameport).

COM порт (COMmunication port, коммуникационный порт) обеспечивает асинхронный обмен и реализуется на микросхемах универсальных асинхронных приемопередатчиков (UATR, Universal Asynchronous Receiver Transmitter). Хотя стандарт RS-232C предусматривает асинхронный и синхронный режимы обмена, COM порт компьютера поддерживает только асинхронный режим. Для реализации синхронного обмена применяются специальные адаптеры, например, SDLC или V.35.

При асинхронной передаче одно из устройств посылает или принимает данные по одному биту (рис. 1.1). Интервалы времени между байтами некритичны,

но времена между отдельными битами в байте очень важны. Сигнал может быть высокого или низкого уровня, что соответствует логическому нулю (SPACE) или единице (MARK). Линия поддерживается в состоянии MARK, когда по ней нет передачи данных. В начале передачи сигнал переключается в 0, отмечая стартовый бит. Затем следуют биты данных (от 5 до 8) в виде набора высоких или низких уровней. Последний бит данных может сопровождаться битом четности, используемым для обнаружения ошибок, а затем в последовательность включаются один или более стоп-битов, которым соответствует высокий уровень. Эти стоп-биты начинают отмеченное состояние (MARK), которое будет сохраняться до тех пор, пока начнется передача следующего байта данных. Число стоп-битов существенно, т.к. они устанавливают минимальное время до передачи следующего стартового бита.

Передатчик и приемник должны использовать один и тот же протокол для этих цепочек битов и должны работать с одной скоростью, измеряемой в битах в сек.



Введем следующие обозначения.

DTE (Data Terminal Equipment)- оконечное оборудование, принимающее или передающее данные. Им может быть компьютер, принтер, плоттер и т.д.

DCE (Data Communication Equipment) – аппаратура канала передачи данных. DCE должно обеспечить соединение DTE с каналом передачи данных. Чаще всего в качестве DCE выступает модем.

Устройства DTE могут быть соединены между собой напрямую без DCE (рис. 1.2) с помощью нуль модемного кабеля, разводка которого показана на рис. 1.3. При необходимости кабель можно изготовить самостоятельно, используя описание сигналов интерфейса RS-232 для разъема DB-9, приведенное в таблице 1. Для лабораторной работы достаточно иметь минимальную конфигурацию кабеля, состоящую из трех проводов (рис. 1.3а).

Рис. 1.1. Полная схема соединения по RS-232

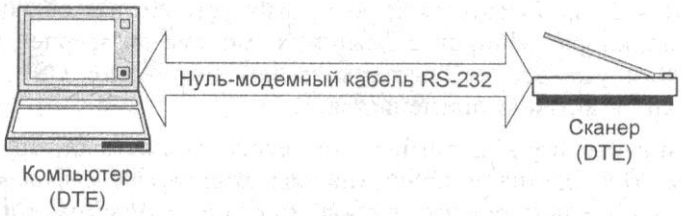


Рис. 1.2. Соединение по RS-232 нуль-модемным кабелем

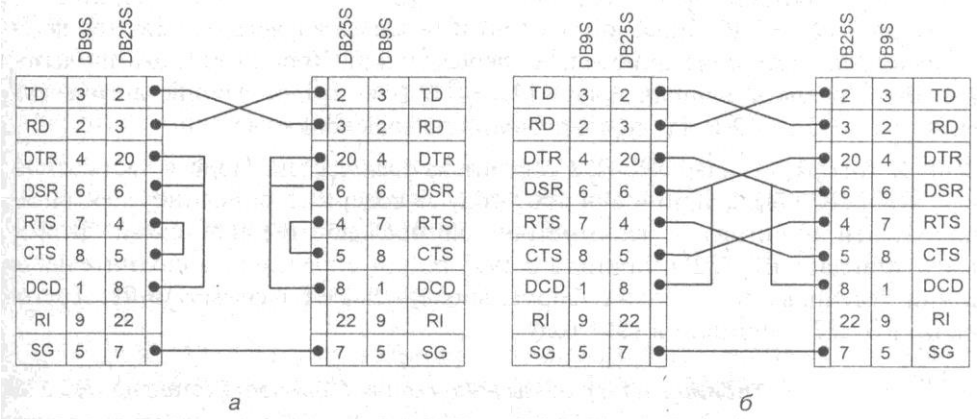


Рис. 1.3. Нуль-модемный кабель:
а — минимальный; б — полный

2.2 Протоколы передачи данных

Под протоколом обмена обычно понимается совокупность договоренностей по обмену данными между передатчиком и приемником. С точки зрения формата передаваемых данных протоколы можно поделить на две категории — строковые и бинарные. В первом случае информация передается обычными ASCII-символами, а во втором — бинарным потоком данных.

Таблица 1

№ провода	Обозначение	Описание (eng)	Описание (rus)	Цвет провода
1	DCD	Carrier Detect	Определение несущей	Чёрный
2	RxD (RD)	Receive Data	Принимаемые данные	Коричневый
3	TxD (TD)	Transmit Data	Передаваемые данные	Красный
4	DTR	Data Terminal Ready	Готовность терминала	Оранжевый
5	GND (SG)	System Ground	Корпус системы	Желтый

6	DSR	Data Set Ready	Готовность DCE	Зелёный
7	RTS	Request to Send	Запрос на отправку Готовность к приему	Синий
8	CTS	Clear to Send	Готовность к передаче	Фиолето-вый
9	RI	Ring Indicator	Индикатор вызова	Белый

Пример пакета данных в формате ASCII приведен в таблице 2. Пакет данных начинается со стартовой последовательности — одного или нескольких символов, обозначающих начало данных. Символы начала и конца посылки не должны появляться внутри блока данных. Получив байты начала посылки, все абоненты настраиваются на получение полного пакета данных. Затем, получив и сравнив номер абонента со своим номером, абонент либо продолжает обработку пакета, либо снова переходит в режим ожидания начала посылки. После получения байта конца посылки абонент проверяет контрольную сумму и обрабатывает полученные данные. Конец посылки также может представляться несколькими символами. Если связь осуществляется между двумя абонентами, поле «номер абонента" может отсутствовать.

Достоинство такого протокола в удобстве отладки: прием и передачу команд и данных можно осуществлять с помощью обычной программы-терминала (например, Telnet). Недостатки - большой объем передаваемых данных и необходимость преобразования двоичных данных в ASCII-строки и обратного преобразования.

Таблица 2

Байт	**Описание**	**Пример 1**	**Пример 2**
Начало посылки	Идентификатор начала посылки. Любой символ (или несколько символов), не встречающийся в символах данных	$	!!!
Номер абонента	Две десятичные цифры номера абонента (дополненные при необходимости нулем). В некоторых случаях могут оговариваться специальные номера абонентов, например, "**", означающая посылку всем абонентам сети.	02	12
Данные	Последовательность цифр данных. Числа с плавающей точкой разделяются знаком".". Части данных могут разделяться пробелом.	011 03.1	123 15.7
Контроль	Контрольная сумма данных. Обычно один или два байта суммы всех байтов данных с переполнением. Байты суммы также передаются в десятичном виде (три десятичные цифры с выравниванием нулями слева).	002	104
Конец посылки	Идентификатор конца посылки. Для ASCII-протоколов обычно символ LR (перевод строки).	$0D	***

Бинарная передача значительно более компактна. Например, для передачи числа с плавающей точкой типа Single или Float требуется 4 байта независимо от

числа знаков после десятичной точки. Так же как в ACSII-протоколе, выбирается некоторый символ для обозначения начала посылки. Завершение ожидания данных производится либо по получению необходимого числа байт, либо по тайм-ауту: принимающее устройство отсчитывает время с получения последнего байта до приема следующего. Если эта пауза превышает некоторый интервал, то посылка считается потерянной и приемный буфер сбрасывается. После получения необходимого числа байт (или символа конца посылки) проверяется правильность полученных данных и производится их обработка.

2.3. Способы предотвращения потери данных

Последовательная передача подразумевает, что все биты, посланные с одной стороны, будут приняты с другой. Если хотя бы часть пакета данных будет потеряна, то это чаще всего означает потерю всего пакета данных.

Причины потери данных при передаче или приеме можно разделить на две категории: аппаратные и программные. К аппаратным причинам относятся помехи на линии, некачественный проводник, несоответствие длины провода и скорости передачи и т. д. Устранение таких причин обычно проводится на аппаратном уровне.

Потеря данных может возникать и при неправильной организации программы. Например, при большой скорости передачи данных может возникнуть ситуация, когда время на обработку полученного пакета данных (например, рисование графика, анализ данных и сохранение их в файл) больше, чем время между получением пакетов. Или организация диалога с пользователем (ожидание ввода с клавиатуры) делает невозможным опрос порта.

2.3.1. Контроль четности

Порт RS-232 реализует аппаратный контроль четности. Пользователь может конфигурировать порт на проверку четного или нечетного паритета или на отсутствие контроля четности. При включенном контроле четности к передаваемому байту добавляется еще один бит таким образом, чтобы количество единиц в этом байте было четным (при четном паритете) или нечетным (при нечетном паритете).

Принимающее устройство производит те же вычисления и сравнивает число единиц в полученном байте. При несовпадении генерируется ошибка четности. Однако, очевидно, что несовпадение четности произойдет только в случае искажения одного бита. При ошибке четности порт может генерировать аппаратное прерывание.

2.3.2. Контрольная сумма

Один из способов повышения достоверности получаемых данных заключается в дополнении посылки контрольной суммой. Контрольная сумма — обычно один, два или четыре байта (в случае необходимости может быть и больше), полученные некоторым преобразованием данных посылки, подсчитываемые перед отправкой и включаемые в посылку. Принимающая программа производит аналогичную операцию над данными и сверяет вычисленную и полученную контрольные суммы. Если помеха изменила один или несколько байт посылки, вероятность совпадения контрольных сумм очень мала.

3. Последовательный порт персонального компьютера

3.1 Конфигурирование и основные принципы работы с COM –портом

ЭВМ может иметь до четырех последовательных портов COM1 — COM4 (в большинстве случаев имеются два порта) с поддержкой на уровне BIOS. Коммуникационные порты занимают в пространстве ввода/вывода по 8 смежных 8-битовых регистров и могут располагаться по стандартным базовым адресам. Порты могут генерировать аппаратные прерывания, конфигурирование портов на системной плате проводится через BIOS Setup.

В табл. 3 представлены базовые адреса COM – портов.

Таблица 3

Порт	**Базовый адрес**	**Генерируемое прерывание**	**Вектор прерывания**	**Адрес в таблице базовых адресов**
COM1	03F8 H	IRQ4	000C H	0400 H
COM2	02F8 H	IRQ3	000B H	0402 H
COM3	03E8 H			0404 H
COM4	02E8 H			0406 H

В таблице 4 приведена структура одного из портов (COM1), где Base-Port - базовый адрес порта.

Таблица 4

Базовый адрес	Относительный адрес	Назначение
03F8	BasePort + 0	Данные и делитель
03F9	BasePort + 1	Разрешение прерываний и делитель
03FA	BasePort + 2	Идентификация прерываний
03FB	BasePort + 3	Управление линией
03FC	BasePort + 4	Управление модемом
03FD	BasePort + 5	Статус линии
03FE	BasePort + 6	Статус модема
03FF	BasePort + 7	Не используется

Делитель – это целое число, которое хранится в двух начальных байтах порта при установке младшего бита порта 03FB в единичное значение, и определяет скорость передачи данных (таблица 5):

делитель = 115200 / *скорость (бит/с*);

Таблица 5

Бит/с	Делитель	03F8	03F9
110	1047	04	17
150	768	03	00
300	384	01	80
600	192	00	C0
1200	96	00	60
2400	48	00	30
4800	24	00	18

Бит/с	Делитель	03F8	03F9
9600	12	00	0C
14 400	8	00	08
19 200	6	00	06
28 800	4	00	04
38 400	3	00	03
57 600	2	00	02
115 200	1	00	01

В процессе начального тестирования BIOS проверяет наличие последовательных портов по стандартным адресам и помещает базовые адреса портов в специальную область данных, находящуюся по адресу 0400H. Нулевое значение адреса является признаком отсутствия порта с данным номером. Обнаруженные порты инициализируются на скорость обмена 2400 бит/с, 7 бит данных с контролем четности и 1 стоп-бит.

Ниже приведен пример программы на Object Pascal, которая читает таблицу базовых адресов и отображает наличие последовательных портов компьютера.

```
Function Byte2Hex(B: Byte):String;
Const hStr : String ='0123456789ABCDEF';
Begin
 Byte2Hex:= hStr[(B div 16)+1]+ hStr[(B mod 16)+1];
End;

Function Long2Hex(B: Longint):String;
Begin
 Long2Hex:= Byte2Hex(B div $100)+ Byte2Hex(B and $0FF);
End;
```

```
{ Возвращает базовый адрес порта с номером PortIndex }
Function GetBaseAdr(PortIndex : Byte) : Word;
Var LowAdr : Word;
Begin
 { вычисляем младшую часть адреса в таблице }
 LowAdr := (PortIndex-1)*2;
 { получаем базовый адрес порта из таблицы }
 GetBaseAdr:= MemW[$0040:LowAdr];
End;

Var
 PortIndex : Byte;
 BaseAdr   : Word;

Begin
 { Опрос базовых адресов 4 портов }
 For PortIndex:= 1 to 4 do begin
  { Получить базовый адрес}
  BaseAdr:= GetBaseAdr(PortIndex);

  { Анализируем базовый адрес }
  If BaseAdr = 0 then
   WriteLn('COM', PortIndex,' не обнаружен')
  Else
   WriteLn('Базовый адрес COM',PortIndex,' равен ', Long2Hex(BaseAdr));
End;
End.
```

В среде C# для чтения таблицы базовых адресов с прямым доступом к памяти можно использовать небезопасный код и указатели [1].

Современные многозадачные операционные системы запрещают прямой доступ ко всем аппаратным ресурсам, включая порты. Поэтому языки программирования высокого уровня для работы с портами предлагают два варианта:

- работа с портом как с обычным файлом;

- использовать специальные библиотеки, оформленные в виде классов, встроенных в среду разработки.

Например, в C++ открыть порт можно следующим образом:

hSerial = ::CreateFile(sPortName,GENERIC_READ | GENERIC_WRITE,0,0,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);

Примерами встроенных классов могут быть SerialPort (C#) или ComPort (Object Pascal). В таблице 6 приведены некоторые методы и свойства этих классов.

Таблица 6

Описание	Имя	
	C#	Object Pascal
Открыть порт	open	Open
Закрыть порт	close	Close
Отправить данные	Write, WriteLine	WriteString,
Считать данные	Read, ReadLine	ReadString

Получить массив имен последовательных портов для текущего компьютера	GetPortNames	GetComName
Возвращает значение True, если порт успешно открыт	IsOpen	Connected
Скорость обмена	BaudRate	BaudRate
Число битов в байте	DataBits	ByteSize
Режим контроля четности	Parity	Parity
Число стоповых битов	Stopbits	Stopbits
Размер входного буфера	ReadBufferSize	InBufSize
Открыть окно настройки свойств порта	-	CommDialog

3.2 Использование виртуальных портов

Современные компьютеры, особенно ноутбуки, часто не имеют COM- порта. В этом случае для подключения внешних COM-устройств можно использовать виртуальные порты, для создания которых существуют различные программы (Virtual Serial Port Kit, VSPD Mobile Phone Edition).

Для выполнения лабораторной работы удобно использовать программу Virtual Null Modem, позволяющую не только создать виртуальные COM- порты, но и соединять их между собой с помощью нуль-модемного кабеля. Кроме того, программа позволяет имитировать помехи произвольного уровня в канале связи между портами. Скачать триальную версию этой программы можно по адресу http://www.virtual-null-modem.com/download.

3.3. Алгоритмы вычисления контрольной суммы

Существует несколько алгоритмов вычисления контрольной суммы (простая контрольная сумма, LRC и CRC).

Простая контрольная сумма вычисляется как исключающее ИЛИ всех информационных байтов посылки. Для вычисления исключающего ИЛИ в языке C (C++) используется оператор ^, а в языке Pascal/Delphi — оператор XOR. Несмотря на примитивность такого метода, вероятность того, что помеха исказит посылку так, что наложение байтов останется неизменным, достаточно мала. А очевидным преимуществом является простота реализации.

Язык C не проверяет выход за границы типа. В Pascal (и, естественно, Delphi тоже) при переполнении вызывается исключение и программа останавливается с ошибкой "Run-time error 201: Range check error". Чтобы этого избежать, следует отключить контроль переполнения при вычислении директивой компилятора {$R-} или программно исключать возможность переполнения.

```
var CS: Byte;
{В Паскале для вычисление контрольной суммы надо отключить контроль переполнения}
Procedure AddCS(b : Byte);
 Begin
  {$R-}
 CS := 0;
 CS :=CS + b;
 {$R+}
 end;

{В Delphi тоже можно отключать переполнение,а можно использовать механизм исключений}
procedure AddCS(b : Byte);
 begin
  Try
    CS:= CS + b;
  Except End;
end;
```

С помощью дополнительной проверки можно обойтись без обработки исключений. Такая конструкция удобна при отладке программы**,** т. к. даже при включенной опции **Останавливать при появлении исключений** (Stop On **Delphi** Exception) программа не будет останавливаться на каждом байте, добавляемом к контрольной сумме.

```
{А с помощью if можно обойтись без обработки исключений }
procedure AddCS(b : byte);
 begin
  if (CS + b) > 256 then
    CS:= CS + b - 256
   else
    CS:= CS + b;
 end;
```

Контрольная сумма LRC (Longiludinal Redundancy Check) - это один байт, который вычисляется передающим устройством и добавляется к концу сообщения. LRC вычисляется сложением последовательности байтов сообщения, отбрасывая все переносы, и затем двойным дополнением результата, т. е. это 8-битовое поле, где каждое новое прибавление символа, приводящее к результату более чем 255, вызывает простое перескакивание через 0. Так как это поле не является 9-битовым, перенос отбрасывается автоматически.

Таким образом, алгоритм генерации LRC выглядит так:

- сложить все байты сообщения, исключая стартовые и конечные символы, складывая их так, чтобы перенос отбрасывался.
- отнять получившееся значение от числа $FF — это является первым дополнением.
- прибавить к получившемуся значению 1 — это второе дополнение.

Ниже приведен пример функции, вычисляющей LRC.

```
Function CalculateLRC(P : PChar; Len : Word) : Byte;
 Var i : Word;
 Begin
  {$R-}
  Result := 0;
   For i:=0 to Len-1 do
     Result:= Result + Byte(Pointer(LongInt(P)+i)^);
   Result := $FF - Result;
   Inc(Result);
    {$R+}
 End;
```

Отметим, что шаги 2 и 3 алгоритма (вычитание $FF и прибавление 1) сводятся к операции Result := - Result, поэтому вычисления можно упростить:

```
Function CalculateLRC(P : PChar; Len : Word) : Byte;
 Var i : Word;
 Begin
  {$R-}
  Result := 0;
   For i:=0 to Len-1 do
     Result:= Result + Byte(Pointer(LongInt(P)+i)^);
   Result := - Result;
    {$R+}
 End;
```

Контрольная сумма CRC16 (Cyclical Redundancy Check) - это 16-разрядная величина, т. е. два байта. В 16-битовый регистр CRC предварительно загружается число SFFFF. Процесс начинается с добавления байтов сообщения к текущему содержимому регистра. Для генерации CRC используются только 8 бит данных. Стартовый и стоповый биты, а также бит паритета, если он используется, не учитываются в CRC.

В процессе генерации CRC каждый 8-битовый символ складывается по исключающему ИЛИ (XOR) с содержимым регистра. Результат сдвигается в направлении младшего бита, с заполнением 0 старшего бита. Младший бит извлекается и проверяется. Если младший бит равен 1, то содержимое регистра складывается с определенной ранее фиксированной величиной по исключающему ИЛИ. Если младший бит равен 0, то исключающее ИЛИ не делается.

Этот процесс повторяется, пока не будет сделано восемь сдвигов. После последнего (восьмого) сдвига, следующий байт складывается с содержимым регистра, и процесс повторяется снова. Финальное содержание регистра после обработки всех байт сообщения и есть контрольная сумма CRC.

Таким образом, алгоритм генерации CRC-16 выглядит так:

- 16-битовый регистр загружается числом SFFFF и используется далее как регистр CRC.
- Первый байт сообщения складывается по исключающему ИЛИ с содер жимым регистра CRC. Результат помещается в регистр CRC.
- Регистр CRC сдвигается вправо (в направлении младшего бита) на один бит, старший бит заполняется 0.
- (Если младший бит 0): Повторяется шаг 3 (сдвиг).
- (Если младший бит 1): Делается операция исключающее ИЛИ регистра CRC и полиномиального числа SA001.
- Шаги 3 и 4 повторяются восемь раз.
- Повторяются шаги 2—5 для следующего сообщения. Это повторяется до

  тех пор, пока все байты сообщения не будут обработаны.

Финальное содержание регистра CRC и есть контрольная сумма.

Приведем пример функции, вычисляющей CRC16.

```
Function CalculateCRC16(P : PChar; Len : Word) : Word;
 Var iByte, i : Word; B: Byte;
 Begin
  {$R-}
   {инициализация}
   Result := $FFFF;
   {цикл по всем байтам буфера}
   For iByte :=0 to Len-1 do
    Begin
      {очередной байт буфера}
      B:=Byte(Pointer(LongInt(P) + iByte)^);
      {вычисление очередного CRC}
      Result:= (Result and $FF00) + (B xor Lo(Result));
      For i := 1 to 8 do
        Begin
         If ((Result and $0001) <> 0) then
            Result :=(Result shr 1) xor $A001
           Else
            Result :=(Result shr 1);
        End;
    End;
  {$R+}
 End;
```

Контрольная сумма CRC32 записывается 4-х байтовым числом и использу-ется для увеличения достоверности данных. Алгоритм вычисления CRC32 практически не отличается от алгоритма вычисления CRC16 за исключением того, что все операции проводятся над словами по 2 байта.

4. Порядок выполнения работы

Обратите внимание: разработка программ для выполнения заданий может проводиться на любом языке программирования.

4.1 Написать и отладить программу, определяющую базовые адреса последовательных портов.

4.2 Написать программу для передачи и приема данных через COM-порт. Основные требования:

- возможность конфигурирования порта (скорость передачи, контроль четности, паритет четности);
- возможность передачи пакета по строковому (ASCII или UTF-8) и по бинарному протоколам в зависимости от выбора пользователя, тип протокола включить в передаваемый пакет данных (например, 0 – строковый, 1 – бинарный);
- возможность передачи символов кириллицы;
- возможность вывода переданных и полученных данных, а также количества переданных и принятых байтов;

4.3 Соединить COM порты компьютера нуль – модемным кабелем. Если в компьютерном классе установлены компьютеры с одним COM портом, то соединить два соседних компьютера. Если COM-порты отсутствуют, то используйте виртуальные порты.

4.4 Запустить 2 копии программы, одну подключить к порту COM1, вторую – к COM2. Провести обмен данными.

4.5 В окне настройки порта COM1 включить контроль четности в состояние «четное», провести обмен данными, пояснить результаты.

4.6 В окне настройки порта COM1 отключить контроль четности и включить скорость передачи данных 9600 бит/с, провести обмен данными и пояснить результаты.

4.7 Написать и отладить функции вычисления простой контрольной суммы, LRC, CRC16 и CRC32. Добавить эти функции к программе, разработанной в п. 4.2, вычисляя контрольные суммы при передаче и при приеме данных, включить контрольную сумму в передаваемый пакет данных.

4.8 Провести тестирование функций для передачи символьной строки согласно Вашего варианта задания (таблица 7).

Таблица 7

№ варианта	Символьная строка посылки	№ варианта	Символьная строка посылки
1	Регистр	8	Данные
2	Вычисление	9	Процесс
3	Редактор	10	Поток
4	Компиляция	11	Задача
5	Компонент	12	Система
6	Windows	13	Функция
7	Программа	14	Процедура

4.10 Изменить один символ в посылке и найти контрольные суммы, результаты занести в протокол.

4.11 Поменять местами два любых символа в посылке и найти контрольные суммы, результаты занести в протокол.

4.12 Передать любое вещественное число типа float, содержащее 5 знаков после запятой, с помощью стокового и бинарного протоколов, результаты занести в протокол.

5. Контрольные вопросы

5.1 Что такое порт, сколько всего портов может иметь компьютер.

5.2 Синхронный и асинхронный способ обмена данными.

5.3 Какие устройства и как можно соединять по протоколу RS-232.

5.4 Протоколы обмена данными.

5.5 Определите эффективность бинарного протокола в сравнении с ASCII протоколом при передаче вещественного числа.

5.6 Способы предотвращения потери данных.

5.7 Что понимается под базовым адресом порта и таблицей базовых адресов.

5.8 Дайте характеристику основным способам контроля достоверности передачи данных.

5.9 Что такое контрольная сумма, для чего и где она используется.

5.10 Простая контрольная сумма, способы ее вычисления.

5.11 Контрольная сумма LRC, способы ее вычисления.

5.12 Контрольная сумма CRC, способы ее вычисления.

Лабораторная работа № 2 Механизмы сообщений.

1. Цель работы

Целью работы является изучение механизма сообщений QNX.

2. Методические указания

Обмен сообщениями – это основная форма межзадачного взаимодействия в QNX Neutrino. В большинстве случаев все другие формы взаимодействий надстраиваются поверх механизма обмена сообщениями. Основная идея заключается в создании простой и надежной службы межзадачного взаимодействия, которую можно оптимизировать для максимальной производительности, используя минимум кода ядра. На основе этой службы можно создать более сложные механизмы межзадачного взаимодействия.

Механизм обмена сообщениями в QNX не совместим со стандартом POSIX.

2.1 Модель синхронных сообщений

Рассмотрим принцип взаимодействия потоков посредством механизма сообщений. Поток, который выполняет передачу сообщения другому потоку, не обязательно принадлежащему тому же процессу, блокируется до тех пор, пока поток-получатель не примет сообщение, не обработает его и не пошлет ответное сообщение. Передача сообщения выполняется с помощью функции MsgSend(), прием сообщения – с помощью функции MsgReceive(), а передача ответного сообщения – с помощью функции MsgReply().

Если поток выполняет функцию MsgReceive() до того, как ему пошлется сообщение, он становится Receive-блокированным до тех пор, пока другой поток не пошлет ему сообщение с помощью MsgSend() .

Схема состояний потока отображена на рисунке 1.

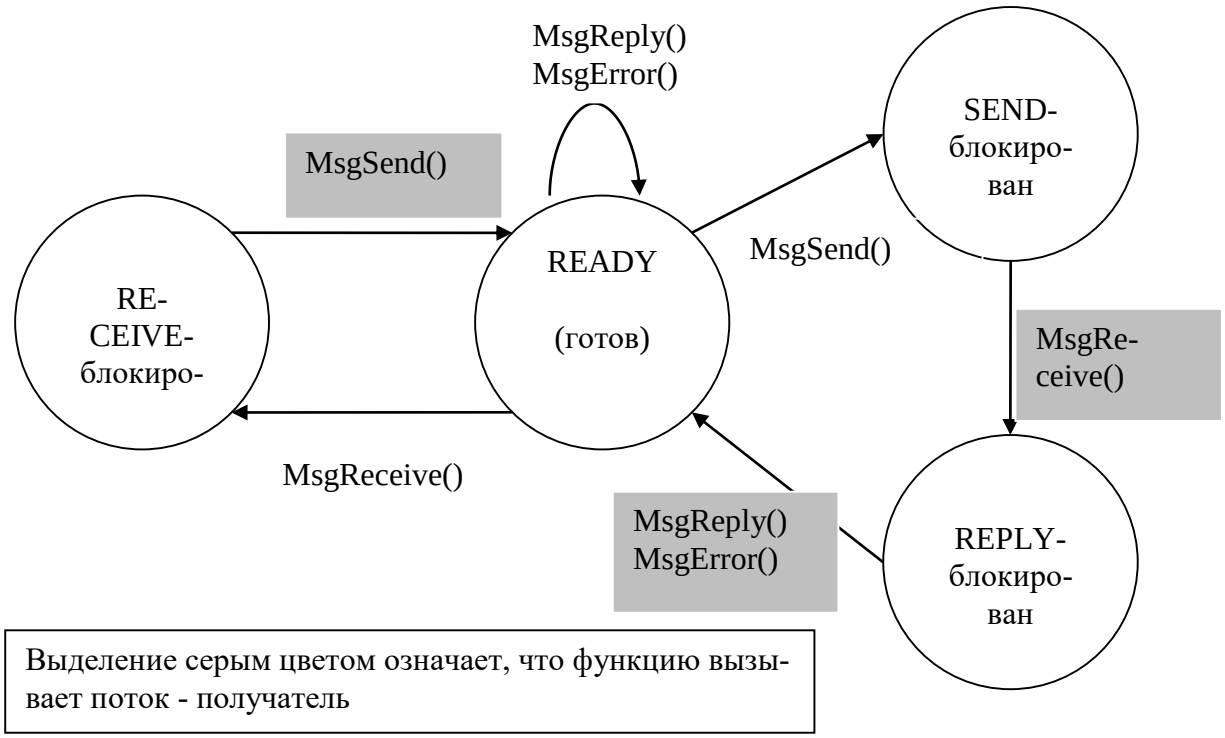


Рис. 1

Таким образом происходит автоматическое блокирование, что обеспечивает синхронизацию отправки и приема сообщений. В результате потоку-отправителю не нужно делать непосредственный запрос на блокирование, чтобы дожидаться ответа, как это потребовалось бы при другой форме межпоточного взаимодействия.

Операции отправки и приема сообщения являются блокирующими и синхронизирующими, тогда как операции отправки ответа и отправки кода ошибки MsgReply() и MsgError() – не блокирующие. Поскольку поток-клиент уже заблокирован для ожидания ответа, дополнительная синхронизация не требуется, поэтому функция MsgReply() не вызывает блокировки. Это позволяет потоку-серверу ответить потоку-клиенту и продолжить свое выполнение, в то время как ОС будет передавать сообщение потоку-клиенту.

На рис. 2 приведена типовая схема взаимодействия двух потоков. Эта схема «клиент-сервер» имеет один существенный недостаток - пока сервер не обработает запрос одного клиента, он не может принимать запросы других клиентов. Для исправления этого применяется многопоточный сервер (на основе пула потоков), когда несколько однотипных потоков сервера параллельно обрабатывают запросы клиентов.

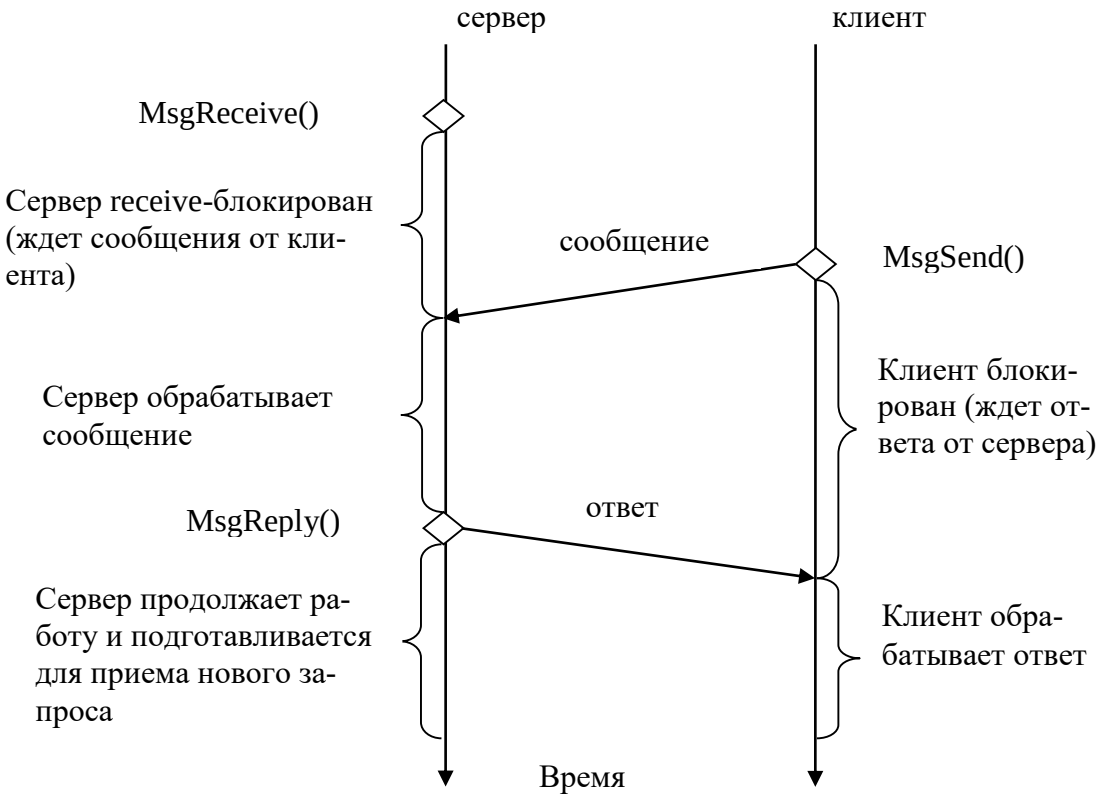


Рис. 2

2.2 Каналы и соединения

В ОС QNX Neutrino сообщения передаются в направлении каналов и соединений, а не напрямую от потока к потоку. Поток, которому нужно получить сообщение (сервер), сначала создает канал, используя функцию ChannelCreate(), а другой поток-отправитель (клиент) должен установить соединение с помощью функции ConnectAttach(), подключившись к этому каналу.

Клиенту для передачи сообщения нужно знать:

- узел сети, на котором выполняется поток-сервер (задается дескриптором узла);
- PID-процесса, содержащего поток-сервер;
- номер канала (Channel ID, CHID).

2.3 Функции для использования синхронных сообщений

Для использования следующих функций к программе должен быть подключен заголовочный файл <sys/neutrino.h>

Ожидание поступления сообщения в канал

*int MsgReceive(int chid, void *msg, int bytes, struct _msg_info *info);*

chid – номер канала, который был указан при его создании (функция ChannelCreate());

msg – указатель на буфер, предназначенный для записи принятого сообщения;

bytes – размер буфера.

info - значение NULL или указатель на структуру _msg_info для сохранения дополнительной информации об сообщении.

Возвращает отрицательное сообщение в случае ошибки. В случае успешного выполнения возвращается параметр *rcvid* (receive identifier – идентификатор полученного сообщения) с помощью которого легко послать ответ на сообщение .

Посылка сообщения в канал.

*int MsgSend(int coid, const void *smsg, int sbytes, void *rmsg, int rbytes);*

coid – номер канала, который был указан в функции соединения ConnectAttach();

smsg – указатель на буфер, предназначенный для отправки;

sbytes –размер буфера, предназначенный для отправки;

rmsg – указатель на буфер для приема ответа;

rbytes – размер буфера для приема ответа;

Возвращает отрицательное сообщение в случае ошибки. В случае успешного выполнения возвращается статус ответа (параметр status функции Msgreply())

Ответ на сообщение

*int MsgReply(int rcvid, int status, const void *msg, int size);*

rcvid – идентификатор получения, возращенный функцией that MsgReceive();

status – статус ответа;

msg – указатель на буфер с ответом;

size – размер буфера для ответа.

Возвращает отрицательное значение в случае ошибки, иначе успешное выполнение.

Разблокирование клиента и установка его переменной errno заданным значением

int MsgError(int rcvid, int error);

rcvid – идентификатор получения, возвращенный функцией MsgReceive();

error – код ошибки, отправляемый клиенту.

Возвращает отрицательное значение в случае ошибки, иначе успешное выполнение.

Создание коммуникационного канала

int ChannelCreate(unsigned flags);

flags – флаги, изменяющие поведение канала.

Возвращает номер созданного канала (CHID) или отрицательное значение в случае ошибки.

Уничтожение коммуникационного канала

int ChannelDestroy(int chid);

chid – идентификатор уничтожаемого канала;

Возвращает отрицательное значение в случае ошибки иначе успех.

Установление соединения между процессом и каналом

int ConnectAttach(uint32_t nd, pid_t pid, int chid, unsigned index, int flags);

nd – дескриптор узла в сети QNET(node descriptor), на котором выполняется процесс которому принадлежит канал (0 - это локальный узел);

pid – идентификатор процесса (process ID), которому принадлежит канал;

chid – идентификатор канала (channel ID), возвращаемый функцией ChannelCreate();

index – наименьшее значение для идентификатора соединения (connection ID);

flags – если флаги содержат _NTO_COF_CLOEXEC, соединение закроется когда процесс вызовет функцию семейства exec*() для старта нового процесса.

Возвращает идентификатор соединения или отрицательное значение в случае ошибки.

Разрыв соединения между процессом и каналом

int ConnectDetach(int coid);

coid – идентификатор разрываемого соединения.

Возвращает отрицательное сообщение в случае ошибки иначе успех.

Из всех перечисленных функций только MsgSend() и MsgReceive() вызывают блокирование потока, остальные – нет.

2.4 Импульсы

Кроме синхронных сообщений, в ОС QNX используются неблокирующие сообщения малого размера. Эти сообщения называются импульсами (pulses).

Основные свойства импульса:

- может перенести 40 бит полезной информации (8-битный код и 32 бита данных);
- является не блокирующим для отправителя;
- может быть получен также, как и сообщение другого типа;
- ставится в очередь, если получатель не заблокирован в ожидании сообщения.

Импульсы являются сообщениями, а поэтому они, как и синхронные сообщения, передаются по соединению клиента к каналу сервера. Для создания канала используется функция *ChannelCreate()*, а для подключения к нему – *ConnectAttach()*.

В отличие от синхронных сообщений, после приема импульса на него нельзя ответить, т.е. вызвать функцию *MsgReply()*.

2.5 Функции для использования импульсов

Отправка импульса

int MsgSendPulse(int coid, int priority, int code, int value);

coid – идентификатор соединения, в которое нужно отправить импульс;

priority – приоритет импульса;

code – 8-битный код импульса, не рекомендуется использовать отрицательные значения (они зарезервированы ядром);

value – 32-битные данные импульса.

Функция возвращает отрицательное значение в случае ошибки.

Прием импульса

Принять импульс можно функцией *MsgReceive()* – она может принимать и импульсы, и обычные сообщения. Если принят импульс, то она возвращает ноль. В случае обычного сообщения, она возвращает идентификатор сообщения. Т.е. отличить импульс от сообщения не составляет труда по возвращаемому значению.

Есть и специальная функция *MsgReceivePulse()*, которая принимает только импульсы:

*int MsgReceivePulse(int chid, void *pulse, int bytes, struct _msg_info * info);*

chid – идентификатор соединения;

pulse – указатель на буфер, для сохранения полученных данных;

info – этот параметр не используется функцией, поэтому следует передавать NULL.

Функция возвращает отрицательное значение в случае ошибки.

В буфер pulse функция *MsgReceivePulse()* записывает структуру _pulse:

```
struct _pulse {
    uint16_t        type;
    uint16_t        subtype;
    int8_t          code;
    uint8_t         zero[3];
    union sigval    value;
    int32_t         scoid;
};
```

Элементы *type* и *subtype* равны нулю. Элемент *scoid* – идентификатор соединения сервера. Элементы *code* и *value* – это 8-битный код и 32 бита данных, т.е. полезная информация импульса, указываемая отправителем, другие поля пользователем не настраиваются. Элемент данных *value* представлен в виде объединения:

```
union sigval
{
  int sival_int;
  void *sival_ptr;
}
```

Т.е. данные могут быть представлены в виде 32 битного числа или указателя на внешние данные.

2.6 Пример взаимодействия на основе механизма сообщений

Ниже приведен пример программы, в которой потоки взаимодействуют по схеме, изображенной на рисунке 2. Поток-сервер основан на функции *server()*, он создает канал с помощью функции *ChannelCreate()* и обрабатывает сообщения в бесконечном цикле. Поток-клиент основан на функции *client()*, причем запускается два экземпляра клиента - один посылает обычное сообщение, другой – импульс.

При приеме сообщения сервер анализирует идентификатор сообщения, возвращаемый функцией *MsgReceive()*. Если идентификатор сообщения равен нулю, то это импульс, и на него не нужно отвечать.

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/neutrino.h>

#define BUFFERSIZE 50

int chid;             // идентификатор канала

void *server()    // поток-сервер
{
        int rcvid;
        int i=0;
        _int8 code;
        int value;
        char receive_buf[BUFFERSIZE], reply_buf[BUFFERSIZE];
        printf("# Server thread:  Channel creating ...");
        // создание канала с опциями по умолчанию и запись в chid номера канала
        chid=ChannelCreate(0);
        if (chid<0)
        {
          perror("Server error");
          exit(EXIT_FAILURE);
        }
        printf("# CHID = %d\n", chid);
        printf("# Server thread: Listen to channel %d\n", chid);
        while (1)        // сервер работает в цикле
        {
         // принимаем сообщение из канала с номером chid в буфер receive_buf
         // в rcvid записывается идентификатор полученного сообщения
                rcvid=MsgReceive(chid, &receive_buf, sizeof(receive_buf), NULL);
```

```
            if (rcvid>0)     // получили обычное сообщение
            {
               printf("# Server thread: message <%s> has received.\n", receive_buf);
               strcpy(reply_buf, "Answer from server");
               printf("# Server thread: answering with <%s> (Status=%d).\n", reply_buf, i);
        // отправляем ответ (буфер reply_buf) по номеру полученного сообщения (rcvid)
        // второй параметр (в данном случае переменная i)
        // статус ответа, обрабатывается клиентом.
               MsgReply(rcvid, i, &reply_buf, sizeof(reply_buf));
                i++;
            }
            if (rcvid==0)   // получили импульс
            {
               code=receive_buf[4]; // 4-ый байт - это код импульса (по структуре _pulse)
               // байты с 8 по 11 - это данные, соберем их в переменную value
               value=receive_buf[11];
               value<<=8; value+=receive_buf[10];
               value<<=8; value+=receive_buf[9];
               value<<=8; value+=receive_buf[8];
               printf("# Server thread: received pulse - code=%d, value=%d.\n", code, value);
            }
        }
}

void *client(void *parametr)// поток-клиент
{
        int coid, status;
        _int8 code;
        int value;
        pid_t PID;
        pthread_t client;
        char send_buf[BUFFERSIZE], reply_buf[BUFFERSIZE];
        PID=getpid();
        client=pthread_self(); // получаем идентификатор потока-клиента
        printf("> Client thread %d: connecting to channel ... ", client);
        // создаем соединение с каналом на текущем узле (0)
        // канал принадлежит процессу с идентификатором PID
        // номер канала - chid
        // наименьшее значение для COID - 0
        // флаги соединения не заданы - 0
        coid=ConnectAttach(0, PID, chid, 0, 0);
        // в coid записан идентификатор соединения или ошибочное значение меньше нуля
        if (coid<0)
        {
```

```
            perror("Client error");
            exit(EXIT_FAILURE);
        }
        printf("COID = %d\n", coid);
        if (client%2==0)        // четные потоки будут отправлять сообщения
        {
            strcpy(send_buf, "It's very simple example");
            printf("> Client thread %d: sending message <%s>.\n", client, send_buf);
            // отправляем сообщение из буфера send_buf в соединение coid
            // ответ принимаем в буфер reply_buf и статус записывается в переменную status
            status=MsgSend(coid, &send_buf, sizeof(send_buf), &reply_buf, sizeof(reply_buf));
            printf("> Client thread %d: I have replied with massage <%s> (status=%d).\n", client, re-
ply_buf, status);
        }
        else
        {  // нечетные потоки будут отправлять импульсы
            code=20;
            value=12345;
            printf("> Client thread %d: sending pulse - code=%d, value=%d.\n", code, value);
            // посылаем импульс в соединение coid
            // приоритет импульса 20
            // код - code, данные - value
            MsgSendPulse(coid, 20, code, value);
        }
        // разрываем соединение coid
        ConnectDetach(coid);
        printf("> Client thread %d:  Good bye.\n", client);
        pthread_exit(NULL);
}



int main()
{
        pthread_t client_tid1, client_tid2;
        printf("Main thread: starting Server & Clients ...\n");
        // создаем потоки сервера и двух клиентов
        pthread_create(NULL, NULL, server, NULL);
        sleep(1);
        pthread_create(&client_tid1, NULL, client, NULL);
        pthread_create(&client_tid2, NULL, client, NULL);
        printf("Main thread: waiting for child threads exiting ...\n");
        // ждем их завершения
        pthread_join(client_tid1, NULL); pthread_join(client_tid2, NULL);
```

```
        printf("Main thread: the end.\n");
        return EXIT_SUCCESS;
}
```

Результат:

```
Main thread: starting Server & Clients ...
# Server thread:  Channel creating ... CHID = 1
# Server thread: Listen to channel 1
> Client thread 3: connecting to channel ... COID = 3
> Client thread 3: sending pulse - code=20, value=12345.
# Server thread: received pulse - code=20, value=12345.
> Client thread 3:  Good bye.
Main thread: waiting for child threads exiting ...
> Client thread 4: connecting to channel ... COID = 3
> Client thread 4: sending message <It's very simple example>.
# Server thread: message <It's very simple example> has received.
# Server thread: answering with <Answer from server> (Status=0).
> Client thread 4: I have replied with massage <Answer from server> (status=0).
> Client thread 4:  Good bye.
Main thread: the end.
```

Как видно, сервер создал канал с идентификатором 1 (CHID). Клиенты-потоки подключаются к каналу через разные соединения с номерами 3 и 4 (идентификаторы COID). Поток с идентификатором 3 послал импульс, а поток с идентификатором 4 – сообщение. Поток 4 получил ответ от сервера, так как послал сообщение. Поток 3, пославший импульс, не ждал ответа, так как это не предусмотрено в импульсах.

3. Порядок выполнения работы

1. Скомпилировать и запустить программу из раздела 2.6, проанализировать результаты.

2. Написать и отладить программу, реализующую задание в соответствии с таблицей 1:

Таблица 1

№ варианта	Задание
1	Клиент посылает серверу строку текста. Сервер возвращает клиенту полученные данные, количество символов передается через статус ответа.
2	Клиент посылает серверу строку текста. Сервер изменяет порядок следования букв в полученном тексте на обратный и отправляет результат клиенту.

3	Клиент посылает серверу строки “TIME”, “DATE”, “RANDOM”, сервер соответственно возвращает время, дату, случайное число.
4	Клиент посылает серверу строку текста. Сервер создает файл с уникальным именем, записывает в него полученные от клиента данные и в качестве результата обработки данных отправляет клиенту имя созданного файла. После получения ответа с сервера клиент распечатывает на экран содержимое указанного сервером файла.
5	Клиент пересылает серверу данные (строку и имя директории). Сервер находит все файлы в заданной директории, содержащие указанную строку, и высылает их имена клиенту.
6	Клиент пересылает серверу имя некоторого файла. Сервер находит файл с указанным именем и пересылает его содержимое клиенту, либо сообщает клиенту, что файл с данным именем не найден.
7	Клиент пересылает серверу имя директории. Сервер возвращает список файлов и поддиректорий данной директории (рекурсивно).
8	Клиенты посылают серверу числа с помощью импульсов в поле данных. Сервер отображает средне арифметическое посланных на данный момент чисел.
9	Реализовать клиент и сервер в разных процессах (сервер и клиент две разные программы). Программа-сервер запускается первой и записывает тройку ND/PID/CHID в файл, программа-клиент считывает эти идентификаторы из файла и с помощью них устанавливает соединение с сервером. Уникальное имя файла задавать константой в тексте программы.

4. Контрольные вопросы

1. Понятие синхронных сообщений QNX, что обозначает термин «синхронный».

2.Опишите схему взаимодействия сервера и клиента при обмене синхронными сообщениями.

3. Имеются ли ограничения на размер сообщений, чем определяется размер передаваемого сообщения.

4. Импульсы: достоинства и недостатки в сравнении с синхронным сообщением.

Лабораторная работа № 3 Синхронизация потоков.

1. Цель работы

Целью работы является изучение механизмов синхронизации.

2. Методические указания

2.1 Общие сведения

В QNX Neutrino реализовано несколько способов синхронизации потоков:

1) взаимоисключающая блокировка (mutual exclusion lock – mutex, "мутекс") – этот механизм обеспечивает исключительный доступ потоков к разделяемым данным. Только один поток в один момент времени может владеть мутексом. Если другие потоки попытаются захватить мутекс, то они становятся мутекс-блокированными.

2) условная переменная (condition variable, или condvar) – предназначена для блокирования потока до тех пор, пока соблюдается некоторое условие. Условные переменные всегда используются с мутекс-блокировкой для определения момента ее снятия;

3) барьер – устанавливает точку для нескольких взаимодействующих потоков, на которой они должны остановиться и дождаться "отставших" потоков. Как только все потоки из контролируемой группы достигли барьера, они разблокируются и могут продолжить исполнение;

4) ждущая блокировка – упрощенная форма совместного использования условной переменной с мутексом. В отличие от прямого применения мутекса и условной переменной имеет некоторые ограничения;

5) блокировка чтения/записи (rwlock) – простая и удобная в использовании "надстройка" над условными переменными и мутексами;

6) семафор – счетчик, используемый для синхронизации.

Все перечисленные способы соответствуют стандарту POSIX, но реально в API QNX (родной интерфейс прикладного уровня QNX) имеются только мутекс, семафор и условная переменная. Остальные механизмы синхронизации реализованы на основе этих трех способов. С помощью комбинации этих механизмов можно строить собственные механизмы синхронизации, если предлагаемые системой механизмы по каким-то причинам не устраивают разработчика [7].

2.2. Мутексы

Назначение мутекса – защита участка кода от совместного выполнения несколькими потоками. Такой участок кода называют критической секцией и обычно в нем происходит модификация общих переменных или обращение к разделяемому ресурсу.

Принцип работы мутекса следующий: поток при входе в критическую секцию обращается к функции захвата *pthread_mutex_lock()*, которая проверяет, захвачен ли уже мутекс. Если да, то поток блокируется до освобождения критической секции. Если же нет, то объект мутекс запоминает, какой поток его захватил и устанавливает признак своего захвата. На выходе из критической секции поток вызывает функцию *pthread_mutex_unlock()*, которая проверяет, что именно этот поток владеет мутексом и разблокирует мутекс. Когда мутекс освобожден, система проверяет, есть ли другие потоки, ждущие его разблокирования; если есть, то один из этих потоков с наивысшим приоритетом захватывает мутекс.

Мутекс хранит информацию о захватившем потоке, поэтому нельзя разблокировать мутекс из другого потока.

В ОС QNX возможен и не описанный в POSIX рекурсивный мутекс. В этом режиме поток, владеющий мутексом, при повторном его захвате не блокируется. Мутекс отмечает в своем счетчике сколько раз он был захвачен, и разблокируется только после равного количества освобождений тем же потоком.

2.3. Функции для использования мутексов

Инициализация мутекса

int pthread_mutex_init(pthread_mutex_t mutex, const pthread_mutexattr_t* attr);*

• mutex – указатель на объект типа pthread_mutex_t (мутекс для инициализации);

• attr – атрибуты мутекса (NULL установит стандартные), которые устанавливаются функцией.

Вместо вызова этой функции статические мутексы можно инициализировать макросами PTHREAD_MUTEX_INITIALIZER или PTHREAD_RMUTEX_ INI-

TIALIZER – первый макрос создает стандартный мутекс, а второй - мутекс с рекурсивным захватом. Следующая строка задает мутекс с параметрами по умолчанию:

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

Простой захват

*int pthread_mutex_lock(pthread_mutex_t *mutex);*

Функция захватывает мутекс, на который ссылается указатель mutex. Если мутекс уже заблокирован другим потоком, то поток, вызвавший функцию, блокируется до освобождения мутекса и после этого захватывает его.

Функция возвращает одно из следующих значений:

EOK – успешно завершение;

EAGAIN –недостаточно системных ресурсов;

EDEADLK – вызывающий поток уже владеет мутексом и мутекс не поддерживает;

рекурсивный захват;

EINVAL – некорректное значение параметра mutex.

Попытка простого захвата

*int pthread_mutex_trylock(pthread_mutex_t *mutex);*

Если мутекс свободен, то функция захватывает его, иначе возвращается значение EBUSY (вызывающий поток не блокируется).

Функция возвращает:

EOK – успешно завершение;

EAGAIN – недостаточно системных ресурсов;

EBUSY – мутекс уже захвачен;

EINVAL – некорректное значение параметра mutex.

Попытка захвата с установкой времени ожидания

*int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abs_timeout);*

Если мутекс свободен, то поток захватывает мутекс, иначе поток блокируется, если текущее время меньше указанного в аргументе abs_timeout. После этого, если мутекс не будет освобожден до времени abs_timeout, то он разблокируется и мутекс не будет захвачен.

Время задается структурой

struct timespec { time_t tv_sec; long tv_nsec; },

где tv_sec – количество секунд с начала 1970 года, tv_nsec – количество добавочных наносекунд.

Функция возвращает следующие параметры:

EOK – успешное завершение;

EAGAIN – недостаточно системных ресурсов для захвата мутекса;

EDEADLK – вызывающий поток уже владеет мутексом и мутекс не поддерживает рекурсивный захват (режим контроля ошибок);

EINVAL –некорректные параметры;

ETIMEDOUT – мутекс не может быть захвачен, поскольку указанный таймаут истек.

Освобождение мутекса

*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

Функция освобождает указанный мутекс. Поток, вызывающий эту функцию, должен быть владельцем мутекса.

Возвращает значения:

EOK – успешное завершение;

EINVAL – некорректное значение параметра mutex;

EPERM – вызвавший поток не является владельцем мутекса.

Разрушение мутекса

*int pthread_mutex_destroy(pthread_mutex_t *mutex);*

Разрушает мутекс. После этого он не может быть использован без предварительного вызова pthread_mutex_init().

Возвращаемые значения:

EOK – успешное завершение;

EBUSY – мутекс захвачен и не может быть разрушен до освобождения;

EINVAL – некорректное значение параметра mutex;

2.4. Пример работы с мутексом

Рассмотрим в качестве примера выполнение потоками списка заданий.

Имеется список заданий, изображенный на рисунке 4.1.

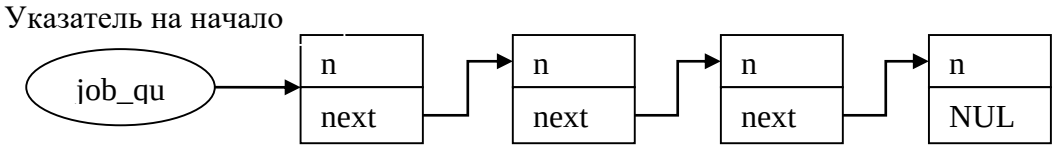


Рис. 3

Каждый элемент списка состоит из атрибута n, описывающего выполняемое задание (вычисление функции), и указателя на следующее задание. Свободный поток считывает параметр n из первого элемента списка, удаляет этот первый элемент и вычисляет задание, зависящее от считанного параметра n.

В данном случае заданием будет вычисление факториала от входного числа n, т.е. требуется вычислить факториал несколько раз от числа n, которое указано в элементах списка.

Этот алгоритм исполняет следующая программа:

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <pthread.h>

#define N 20 // число заданий

// структура-элемент списка
struct job
{
        int n;                  // параметр задания
        struct job *next;       // указатель на следующее задание
};

struct job *job_queue;          // список заданий

// функция выполнения задания
void process_job(struct job *the_job)
{
        int i, M=0;
        long factorial=1;
```

```
        printf("Thread<%d> run jobs with parametr: %d\n", pthread_self(), the_job->n);
// здесь проводятся вычисления
// в данном случае вычисление факториала из числа
                // указанного в задании (число n).
        M=the_job->n;
        for(i=2; i<=M; i++)
        {
                factorial*=i;
        }
        printf("Thread<%d>: %d!=%ld\n", pthread_self(), M, factorial);
}

// функция потока, выполняющего задания
void *thread_function(void *arg)
{
        while(job_queue!=NULL)
        {
                // пока есть элементы в списке заданий
                // выполняем задание и удаляем его из списка
                struct job *next_job=job_queue;
                job_queue=job_queue->next;
                process_job(next_job);
                free(next_job);
        }
        return NULL;
}

// функция-генератор заданий
void *jobs_producer(void *arg)
{
        int i;
        struct job *point;
        for(i=N; i>0; i--)
        {
                // создаем список из N заданий
                point=(struct job*)malloc(sizeof(struct job));
                point->n=rand()%12;
                point->next=job_queue;
                job_queue=point;
        }
        return NULL;
}

int main(int argc, char *argv[])
{
        pthread_t first_tid, second_tid;
        // генерируем задания
        jobs_producer(NULL);
        // и создаем два потока для их выполнения
        pthread_create(&first_tid, NULL, thread_function, NULL);
        pthread_create(&second_tid, NULL, thread_function, NULL);

        // ждем завершения потоков, которые выполняют задания
        pthread_join(first_tid, NULL);
        pthread_join(second_tid, NULL);
```

```
    return EXIT_SUCCESS;
}
```

Сначала выполняется функция jobs_producer(), которая создает список заданий. После того, как список создан, запускаются два потока, которые выполняют задания в списке и удаляют их. Когда в очереди не останется заданий, потоки завершатся и программа прекратит выполнение.

Результат выполнения:

```
Thread<2> run jobs with parametr: 6
Thread<2>: 6!=720
Thread<2> run jobs with parametr: 3
Thread<2>: 3!=6
…
Thread<3> run jobs with parametr: 9
Thread<3>: 9!=362880
```

Видно, что два потока с идентификаторами 2 и 3 выполняют задания по вычислению факториалов, причем сначала задания выполнял поток 2, потом поток 3. В зависимости от текущей нагрузки системы, входных параметров и других факторов последовательность сообщений от двух потоков может быть другой.

При выполнении данного примера могут возникать ошибки, связанные с тем, что оба потока конкурентно пытаются завладеть одним ресурсом – списком заданий. Например, если в очереди останется один элемент, то оба потока пройдут условия на не пустоту списка заданий и будут выполнять одно и то же задание, причем один поток удалит задание из списка, а второй поток будет пытаться освобождать нулевой указатель, что приведет к ошибке исполнения. Возможен также пропуск некоторых заданий или выполнение их дважды.

Данный эффект называется гонкой за ресурсами. Чтоб устранить эту проблему, нужно сделать операции по работе со списком атомарными с помощью критических секций.

Добавим после секции директив #include объявление мутекса:

```
…….
// создаем мутекс, для того чтоб потоки не работали
// со списком одновременно
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
…….
```

И перепишем функцию **thread_function():**

```
void *thread_function(void *arg)
```

```
{
        while(1)
        {
                struct job *next_job;
                // захватываем мутекс
                pthread_mutex_lock(&mutex);
                // в критической секции монопольно работаем со списком заданий
                if (job_queue==NULL)
                        next_job=NULL;
                else
                {
                        next_job=job_queue;
                        // переводим указатель на следующее задание
                        job_queue=job_queue->next;
                }
                pthread_mutex_unlock(&mutex);
                // на выходе из критической секции
                // получаем указатель на текущее задание (next_job)
                if (next_job==NULL)
                        break;
                process_job(next_job);
                free(next_job);
        }
        return NULL;
}
```

Теперь операции над списком заданий производится в критической секции, обрамленной вызовами функций **pthread_mutex_lock()** и **pthread_mutex_unlock()**, что гарантирует атомарность выполнения операции над списком.

2.5. Семафоры

Семафор – объект, над которым можно провести две атомарных операции - инкремент и декремент внутреннего счетчика при условии, что внутренний счетчик не может принимать значения меньше нуля. Если некий поток пытается уменьшить на единицу значение внутреннего счетчика семафора, значение которого уже равно нулю, то этот поток блокируется до тех пор, пока внутренний счетчик семафора не примет значения 1 или больше (посредством воздействия на него других потоков), чтобы указанный поток мог осуществить декремент этого значения.

Семафор не может знать потока захватившего его, поскольку у семафора в принципе не может быть владельца. Поэтому следует тщательно проектировать потоки, чтобы избежать ситуации взаимной блокировки потоков на семафоре, при которой семафор уже не будет разблокирован (deadlock).

QNX поддерживает два типа семафоров – неименованные и именованные. Последние позволяют связывать любые процессы системы (даже выполняемые на различных узлах сети) путем назначения семафору имени в файловой системе. По той же причине именованные семафоры медленнее неименованных.

2.6. Функции для работы с семафором

Инициализация и уничтожение семафора

Создание неименованного семафора семафора реализует функция

*int sem_init(sem_t *sem, int pshared, unsigned value);*

• sem – указатель на объект типа sem_t (семафор);

• pshared – не ноль, если требуется предоставление семафора процессам посредством совместной памяти;

• value – инициализатор счетчика семафора(значение 0 инициализирует семафор блокированным). Это значение не должно превышать константу SEM_VALUE_MAX.

Разрушение неименованного семафора проводится функцией

*int sem_destroy(sem_t *sem)*

Обе функции возвращают 0 в случае успешного выполнения и отрицательное значение при возникновении ошибки.

Для создания именованного семафора используется функция:

*sem_t *sem_open(const char *sem_name, int oflags, ...);*

• sem_name – имя семафора;

• oflags – флаги. Флаг O_CREAT указывает на то, что если семафора с таким именем нет, то он будет создан. Совместный флаг O_CREAT|O_EXCL требует, чтоб функция sem_open() завершилась с ошибкой, если семафор с таким именем уже существует (действие флагов, как в функции открытия файлов open()).

Если указан флаг O_CREAT, то нужно указать еще два аргумента:

• mode_t mode – права доступа семафора (как и у прав файлов). Константы S_IRWXG, S_IRWXO, S_IRWXU задают полный набор прав (чтение, запись, выполнение) соответственно для группы, для остальных, для владельца.

• unsigned int value – инициализатор счетчика семафора.

Функция возвращает указатель на семафор или отрицательное значение в случае ошибки.

Закрывается именованный семафор функцией

*int sem_close(sem_t *sem),*

единственным аргументом которой является указатель на семафор.

Чтобы удалить именованный семафор, нужно вызвать функцию

*int sem_unlink(const char *sem_name),*

которая принимает имя семафора.

Примечание: не стоит смешивать функции создания и уничтожения именованных и неименованных семафоров при операциях над одним семафором – это приведет к ошибке исполнения.

Блокировка семафора

Для семафора определены три модификации операции блокировки:

*int sem_wait(sem_t *sem);*

*int sem_trywait(sem_t *sem);*

*int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);*

Функция простого ожидания **sem_wait()** пытается выполнить декремент счетчика семафора. Если исходное значение счетчика было больше или равно 1, то функция после выполнения декремента возвращает управление в вызвавший код. Если же значение внутреннего счетчика семафора равнялось 0, то выполнение вызвавшего функцию потока блокируется до тех пор, пока какой-либо другой поток не увеличит значение счетчика семафора. После того, как значение счетчика увеличивается, блокированный поток в положенное ему время получает управление, выполняет декремент счётчика семафора, и возвращает управление.

Функция ожидания с проверкой семафора – **sem_trywait()** до выполнения операции над семафором проверяет значение счетчика и, если он больше нуля, то выполняет его декремент, если же он равен нулю – возвращает управление потоку, не блокируя его для ожидания доступности семафора (захват семафора в этом случае не произошёл, о чём извещает отрицательный код возврата и переменная errno равная EAGAIN).

Функция ожидания с тайм-аутом – **sem_timedwait()** ожидает возможности декремента счетчика семафора (блокируется на ожидании) до момента времени abs_timeout.

Разблокирование семафора

*int sem_post(sem_t *sem)*

Эта операция увеличивает на единицу внутренний счетчик семафора – в этом она диаметрально противоположна операции блокирования. Если перед этим значение счетчика семафора было равно 0, то один из потоков, ожидавших разблокирования семафора переходит в состояние готовности.

Функция sem_post() сообщает о следующих ошибках:

EINVAL – неверный дескриптор семафора sem;

ENOSYS – функция sem_post() не поддерживается системой.

Получение счетчика семафора

*int sem_getvalue(sem_t *sem, int *value);*

Функция записывает в переменную, на которую ссылается указатель value, значение счетчика семафора. Используется в основном для отладки.

2.7. Пример работы с семафором

Модифицируем приложение, описанное в разделе 2.4. Сделаем так, чтобы список заданий пополнялся динамически во время работы двух потоков-исполнителей.

Тогда возникнет проблема, связанная с тем, что потоки-исполнители при отсутствии заданий в списке прекращают выполнение, т.е. они могут выполнить все задания и завершиться. В этом случае, если в очередь поступит новое задание, то исполнять его уже некому.

Для блокирования потоков в случае отсутствия в очереди заданий используем семафор. Внутренний счетчик этого семафора будет определяться количеством заданий в списке. Поток, который добавляет задания, вызывает функцию sem_post(), тем самым увеличивает на единицу счетчик. Поток-исполнитель, который захотел выполнить задание, вызывает функцию sem_wait(), что приводит к уменьшению

счетчика. Если значение счетчика семафора равно нулю, т.е. нет заданий для выполнения, то поток-исполнитель, вызвав функцию sem_wait(), заблокируется и будет разблокирован, когда появятся задания.

С учетом вышесказанного программа примет следующий вид:

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>

struct job
{
	int n;
	struct job *next;
};

struct job *job_queue;

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

// объявляем семафор для блокирования потоков-исполнителей,
// если нет заданий в списке
sem_t sem;

// функция выполнения задания
void process_job(struct job *the_job)
{
	int i, M=0;
	long factorial=1;
	printf("Thread<%d> run jobs with parametr: %d\n", pthread_self(), the_job->n);
// здесь проводятся вычисления
// в данном случае вычисление факториала из числа, указанного в задании (число n).
	M=the_job->n;
	for(i=2; i<=M; i++)
	{
		factorial*=i;
	}
	printf("Thread<%d>: %d!=%ld\n", pthread_self(), M, factorial);
}

void *thread_function(void *arg)
{
	while(1)
	{
		struct job *next_job;
		// уменьшаем на единицу счетчик семафора.
		// если счетчик равен нулю, то поток
		// будет ждать увелечения значения счетчика
		sem_wait(&sem);
		pthread_mutex_lock(&mutex);
		// входим в критическую секцию
```

```
                // и монопольно действуем со списком заданий -
                // удаляем одно задание из списка
                next_job=job_queue;
                job_queue=job_queue->next;
                pthread_mutex_unlock(&mutex);
                process_job(next_job);
                free(next_job);
        }
        return NULL;
}
// функция-генератор заданий
// задания генерируется во время работы
void *jobs_producer(void *arg)
{
        struct job *point;
        while(1)
        {
                // создаем элемент-задание
                point=(struct job*)malloc(sizeof(struct job));
                point->n=rand()%12;
                // входим в критическую секцию
                // для записи задания в список
                pthread_mutex_lock(&mutex);
                point->next=job_queue;
                job_queue=point;
                // увеличиваем на единицу счетчик семафора,
                // тем самым сообщая, что появилось задание
                sem_post(&sem);
                pthread_mutex_unlock(&mutex);
                sleep(1);       // задания генерируется с задержкой
        }
        return NULL;
}

int main(int argc, char *argv[])
{
        // в начале список заданий пуст
        job_queue=NULL;
        // инициализируем счетчик семафора значением ноль
        sem_init(&sem, 0, 0);
        pthread_create(NULL, NULL, jobs_producer, NULL);
        pthread_create(NULL, NULL, thread_function, NULL);
        pthread_create(NULL, NULL, thread_function, NULL);

        // подождем 10 секунд и завершим приложение
        sleep(10);
        return EXIT_SUCCESS;
}
```

Результат выполнения

```
Thread<3> run jobs with parametr: 2
Thread<3>: 2!=2
Thread<4> run jobs with parametr: 10
Thread<4>: 10!=3628800
```

Thread<3> run jobs with parametr: 9
Thread<3>: 9!=362880
…

2.8. Условные переменные

Условная переменная (condvar – сокр. от condition variable) используется для блокирования потока по какому-либо условию во время выполнения критической секции, т.е. условная переменная используется совместно с мутексом для проверки условия. Условие может быть сколь угодно сложным и не зависеть от условной переменной.

Условная переменная заставляет поток ждать внутри критической секции некоторого условия, при этом мутекс освобождается. Когда условие наступило, поток пытается захватить мутекс. Этот механизм отображен на рисунке 4.

2.9. Функции для работы с условной переменной

Инициализация условной переменной

*int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);*

Здесь

- cond – указатель на переменную типа pthread_cond_t, которую нужно инициализировать;
- attr – NULL или указатель на объект типа pthread_condattr_tthat, в котором записаны дополнительные атрибуты условной переменной.

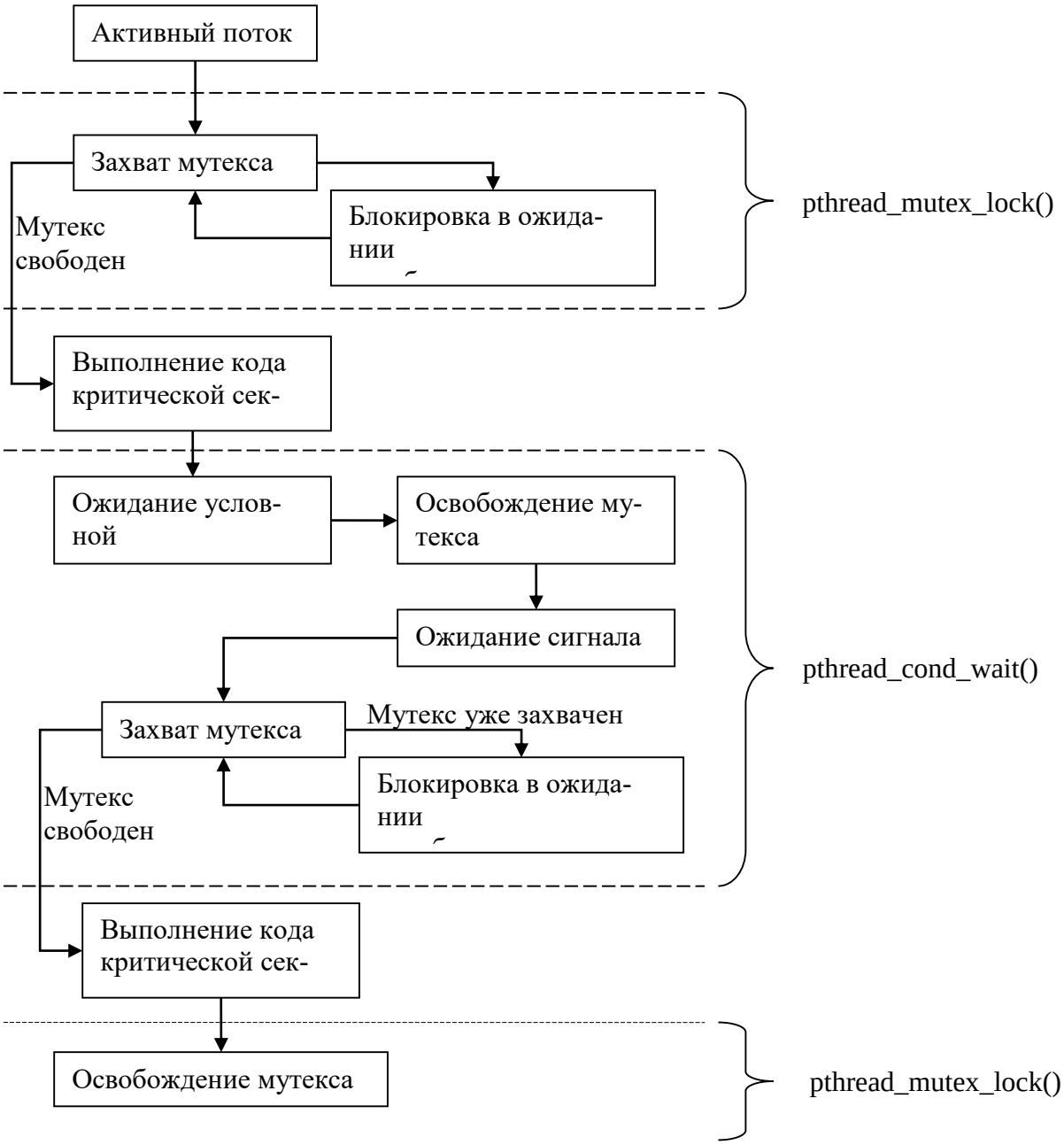


Рис. 4

Функция возвращает:

EOK – успешное завершение.

EAGAIN – Нет свободных системных объектов синхронизации.

EBUSY – Переменная cond уже инициализирована и не разрушалась.

EFAULT – ошибка доступа ядра к объектам cond или attr.

EINVAL – неправильное значение переменной cond.

Для статической инициализации условной переменной можно воспользоваться макросом PTHREAD_COND_INITIALIZER:

pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

Функции ожидания условия

*int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);*

*int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);*

Вызов функции **pthread_cond_wait()** блокирует вызвавший поток на условной переменной cond и разблокирует мутекс mutex. Поток блокируется до тех пор, пока другой поток не вызовет функцию разблокирования на условной переменной cond **pthread_cond_signal()** или **pthread_cond_broadcast().** Мутекс mutex должен быть захвачен потоком до вызова функции. Поток, блокированный на условной переменной, может быть разблокирован также приходом сигнала или вызовом завершения потока. В любом случае при разблокировании потока и выходе из функции ожидания условия поток вновь захватывает мутекс mutex.

Поведение функции **pthread_cond_timedwait()** полностью идентично варианту обычного ожидания за тем исключением, что ожидание может завершится также при наступлении времени, переданного параметром abstime.

Обе функции возвращают:

EOK – успешное завершение ожидания либо ожидание прервано сигналом;

EAGAIN – недостаток системных ресурсов;

EFAULT – произошла ошибка при попытке обращения указателям cond или mutex;

EINVAL – возвращается в случае следующих ситуаций:

- не инициализированы переменные cond или mutex;
- попытка использования переменной cond для нескольких мутексов;
- вызвавший поток не владеет указанным мутексом.

Для функции **pthread_cond_timedwait()** есть еще один код возврата:

ETIMEDOUT – завершение функции по наступлению времени, указанного в abstime.

Разблокировка по выполнению условия

*int pthread_cond_signal(pthread_cond_t *cond);*

*int pthread_cond_broadcast(pthread_cond_t *cond);*

Чтобы разблокировать поток, ожидающий выполнения условия (блокирован на условной переменной), нужно выполнить в другом потоке функции **pthread_cond_signal()** или **pthread_cond_broadcast().** Первая разблокирует самый приоритетный поток из блокированных на условной переменной, вторая функция разблокирует все эти потоки. Но во втором случае только один поток захватит мутекс (самый приоритетный), остальные же перейдут в мутекс-блокированное состояние.

Обе функции возвращают:

EOK – успешное завершение.

EFAULT – произошла ошибка при попытке обращения к указателям cond или mutex.

EINVAL – не инициализирована переменная cond.

Уничтожение условной переменной

*int pthread_cond_destroy(pthread_cond_t *cond);*

Функция возвращает:

EOK – успешное завершение.

EBUSY – другой поток блокирован в данный момент на условной переменной cond.

EINVAL – не инициализирована переменная cond.

2.10. Типичное применение условной переменной

```
pthread_mutex_lock(&mutex);
. . .
while(!condition)
{
        pthread_cond_wait(&condvar, &mutex);
}
. . .
pthread_mutex_unlock(&mutex);
```

В этом примере захват мутекса происходит до проверки условия, поэтому проверяемое условие применяется только к текущему потоку. Пока условие выполняется, поток блокируется до тех пор, пока какой-нибудь другой поток не выполнит операцию единичной или множественной разблокировки по условной переменной. Цикл while требуется по двум причинам:

1) Стандарты POSIX не гарантируют отсутствие ложных пробуждений.
2) Если другой поток изменит условие, то необходимо заново выполнить его проверку.

2.11. Пример работы с условной переменной

В качестве примера рассмотрим взаимодействия двух потоков. Один получает данные с устройства (производитель данных), другой их обрабатывает (потребитель данных).

В качестве условия для условной переменной будет выступать готовность данных. Сначала флаг готовности установлен на отсутствие данных, и потребитель останавливается на условной переменной, производитель же проходит проверку условия на неготовность данных и получает их с устройства. Далее производитель устанавливает флаг на готовность данных и функцией **pthread_signal()** разблокирует потребителя, после чего блокируется на условной переменной. Потребитель проходит проверку на готовность данных, обрабатывает данные, устанавливает флаг на отсутствие данных и далее разблокирует производителя.

Т.е. производитель и потребитель выполняются по очереди - сначала производитель готовит данные, затем потребитель их обрабатывает. Флаг готовности данных чередуется – данных нет, данные есть, данных нет, данные есть и т.д.

Этот алгоритм представлен в коде ниже (реально действий с данными не происходит, чтобы не усложнять код):

```
#include <stdio.h>
#include <pthread.h>

// флаг готовности данных
int data_ready=0;
//статически инициализируем мутекс и условную переменную
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar=PTHREAD_COND_INITIALIZER;

// этот поток обрабатывает данные - потребитель
void *consumer(void *notused)
{
        printf("In consumer thread...\n");
        while(1)
        {
                pthread_mutex_lock(&mutex);
                while(!data_ready)
                {
                        // если нет данных, то ждем, пока они появятся
                        pthread_cond_wait(&condvar, &mutex);
```

```
                }
                printf("consumer:  got data from producer\n");
                // здесь может быть обработка данных
                // это имитируется функцией sleep()
                sleep(1);
                // устанавливаем флаг, что нет необработанных данных
                data_ready=0;
                // разблокируем производителя данных
                pthread_cond_signal(&condvar);
                pthread_mutex_unlock(&mutex);
        }
}

// этот поток получает данные - производитель
void *producer(void *notused)
{
        printf("In producer thread...\n");
        while(1)
        {
                // получаем данные из устройства
                // это имитируется функцией sleep()
                sleep(1);
                printf("producer:  got data from h/w\n");
                pthread_mutex_lock(&mutex);
                while(data_ready)
                {
                        // если есть данные, то ждем, пока они обработаются
                        pthread_cond_wait(&condvar, &mutex);
                }
                // устанавливаем флаг, что есть необработанные данные
                data_ready=1;
                // разблокируем потребителя данных
                pthread_cond_signal(&condvar);
                pthread_mutex_unlock(&mutex);
        }
}
int main()
{
        printf("Starting consumer/producer example...\n");

        // создаем потоки производителя и потребителя данных
        pthread_create(NULL, NULL, producer, NULL);
        pthread_create(NULL, NULL, consumer, NULL);

        // подождем немного и завершим приложение
        sleep(10);
        return 0;
}
```

Результат выполнения:

```
Starting consumer/producer example...
In producer thread...
In consumer thread...
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
```

```
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
…
```

Потоки потребителя и производителя ждут выполнения условия готовности и неготовности данных соответственно. И сообщают друг другу о выполнение условия с помощью функции **pthread_cond_signal().**

3. Порядок выполнения работы

1. Скомпилировать и выполнить примеры программ. Проанализировать результат выполнения.

2. Написать программу, реализующую задание в соответствии с таблицей 2:

Таблица 2

№ варианта	**Задание**
1	Два потока выполняют циклически запись в файл текущего времени и некоторого строки-сообщения. Синхронизацию записи в файл между ними произвести с помощью мутекса.
2	Два процесса (две разные программы) выполняют циклически запись в файл текущего времени и некоторого стороки-сообщения. Синхронизацию записи в файл между ними произвести с помощью именованного бинарного семафора (значение счетчика равно нулю – файл занят, единице – свободен).
3	На основе примера очереди заданий сделать подсчет функции f(x)=Exp(x**2/2)/Sqrt(x!) для списка аргументов, которые задаются в файле.
4	На основе механизма условной переменной, разработать приложение подсчета функции f(x)=Exp(x**2/2)/Sqrt(x!). Один поток читает следующий аргумент из файла в общую переменную, второй поток подсчитывает результат функции для аргумента.
5	На основе механизма условной переменной, разработать приложение подсчета простой контрольной суммы файла. Один поток читает порциями данные из файла в общий буфер, второй поток подсчитывает контрольную сумму.
6	Задание аналогично варианту 5, но подсчет контрольной суммы LRC.
7	Задание аналогично варианту 5, но подсчет контрольной суммы CRC16.
8	Задание аналогично варианту 5, но подсчет контрольной суммы CRC32.

4. Контрольные вопросы

1. Назовите основные способы синхронизации потоков и их характеристики.
2. Сравните между собой мутекс и семафор.
3. Когда необходимо использовать для синхронизации условную переменную.
4. Функции управления объектами синхронизации.

Лабораторная работа № 4. Таймеры.

1. Цель работы

Цель работы – изучение механизма таймеров.

2. Методические указания

2.1. Общие сведения

Фиксация временных интервалов и хронометраж выполнения участков кода приложений для ОС реального времени существенно более критичны, чем для операционных систем общего назначения. ОС QNX Neutrino обеспечивает полный набор таймеров стандарта POSIX, которые являются очень удобным инструментом ядра, т.к. их можно быстро создать и ими легко управлять.

Время в ОС QNX дискретно. Единичный момент времени – это такт системного времени. Начальная (после загрузки системы) длительность тика в ОС QNX 6 для процессоров с частотой больше 40МГц составляет 1мс. Таймер отсчитывается ядром по системным тикам, поэтому не стоит ждать от таймера точности большей, чем системный тик.

Таймер может задаваться относительным или абсолютным временем. Относительный таймер срабатывает через заданный интервал времени, т.е. отсчет проводится от текущего времени, а абсолютный таймер срабатывает в заданный момент времени.

Таймер может работать в однократном или циклическом режиме. Периодический таймер – это таймер, который срабатывает периодически, уведомляя поток о том, что истек некоторый интервал времени. Однократный таймер – это таймер, который срабатывает только один раз.

Уведомление о наступлении тайм-аута таймера передаются тремя способами:

- послать импульс;
- послать сигнал;
- создать поток (при наступлении события запустится поток, который будет выполнять заданную функцию).

Последний способ нужно использовать с осторожностью, т.к. он уступает по эффективности импульсам и при частом срабатывании таймера созданные потоки могут исчерпать ресурсы системы.

2.2. Работа с таймерами

Создание таймера

Для создания таймера используется функция **timer_create().**

```
#include <signal.h>
#include <time.h>
int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timerid);
```

Здесь

- clock_id – тип временного базиса (для QNX применяется CLOCK_REALTIME);
- evp – структура типа sigevent, задающая тип события, который таймер должен сгенерировать при срабатывании;
- timerid – указатель на переменную типа timer_t, в которую запишется идентификатор таймера.

Структура типа sigevent содержит поля, описывающие атрибуты события при срабатывании таймера. Их можно задавать непосредственным присваиванием значений или с помощью следующих макросов:

а) уведомление импульсом

Макрос *SIGEV_PULSE_INIT(&event, coid, priority, code, value)* установит структуру event на посылку импульса в канал coid с приоритетом priority, код и данные в импульсе будут равны code и value соответственно. Если в качестве priority послать константу SIGEV_PULSE_PRIO_INHERIT, то приоритет принимающего потока не будет изменен.

б) уведомление сигналом

Макрос *SIGEV_SIGNAL_INIT(&event, signal)* установит структуру event на посылку процессу обычного сигнала с номером signal.

Макрос *SIGEV_SIGNAL_CODE_INIT(&event, signal, value, code)* задает посылку процессу сигнала с номером signal, который (сигнал) имеет 32-битное поле данных и 8-битный код.

Макрос *IGEV_SIGNAL_THREAD_INIT(&event, signal, value, code)* делает тоже самое, что и предыдущий макрос, но сигнал будет отослан потоку вызвавшему функцию timer_settime().

в) создание потока

Макрос *SIGEV_THREAD_INIT(eventp, func, value, attributes)* задает при срабатывании таймера создание потока с атрибутами attributes, который вызовет функцию func, которой будет передано значение value.

Включение таймера

После того как таймер создан, нужно задать время его срабатывания функцией **timer_settime().**

#include <time.h>
*int timer_settime(timer_t timerid, int flags, struct itimerspec *value, struct itimerspec *ovalue);*
Здесь

- timerid – идентификатор таймера;
- flags – устанавливает способ задание времени срабатывания таймера (относительное или абсолютное). Если передать ноль, то параметр value трактуется как смещение от текущего времени. Если передать константу TIMER_ABSTIME, то таймер сработает в момент времени value (количество секунд и наносекунд с начала 1970 года).
- value – время срабатывания таймера;
- ovalue – NULL или указатель на структуру itimerspec, куда будут записаны старые значения время срабатывания таймера, если таймер еще не сработал.

Параметр value (время срабатывания таймера) задается структурой itimerspec:

```
struct itimerspec
{
        struct timespec it_value;
        struct timespec it_interval;
}
```

Параметр it_value задает интервал времени от настоящего момента или собственно время срабатывания в зависимости от режима (относительный или абсолютный). Установка этого параметра в ноль сделает таймер не рабочим, т.е. выключает его.

Параметр it_interval задает интервал времени, по истечении которого таймер будет срабатывать периодически. Установка этого параметра в ноль приведет к тому, что таймер сработает только один раз (однократный таймер).

Оба параметра it_value и it_interval являются структурами типа timespec:

```
struct timespec
{
        long tv_sec;
        long tv_nsec;
}
```

Элемент tv_sec задает количество секунд, а tv_nsec – количество наносекунд.

Удаление таймера

После завершения процесса, создавшего таймер, сам таймер будет удален. Но если таймер уже не будет использоваться, его можно удалить с помощью функции **timer_delete().**

```
#include <time.h>
int timer_delete(timer_t timerid);
```

Функция удаляет таймер с идентификатором timerid, тем самым таймер удаляется из списка активных таймеров системы.

Следующее приложение демонстрирует работу с таймером. Главный поток создает канал и сам же к нему подключается. Далее создается многократный таймер, уведомляющий импульсом. Главный поток принимает импульсы, которые сообщают, что прошел очередной интервал таймера и выводит сообщение об этом на экран.

```
#include <stdio.h>
#include <time.h>
#include <sys/netmgr.h>
#include <sys/neutrino.h>

// Задаем код импульса
// он должен быть между _PULSE_CODE_MINAVAIL и _PULSE_CODE_MAXAVAIL
#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL

int main()
{
        int i;
        struct sigevent event;
        struct itimerspec itime;
        timer_t timer_id;
        int chid, coid;
```

```
        struct _pulse impulse;

        // создаем канал
        chid=ChannelCreate(0);
        // и сами к нему соединияемся
        coid=ConnectAttach(ND_LOCAL_NODE, 0, chid, 0, 0);

        // устанавливаем структуру event на уведомление
        //   импульсом с кодом MY_PULSE_CODE в канал coid.
        // импульс передается с приоритетом текущего потока,
        // который получен с помощью функции getprio(0).
        // последний аргумент равен нулю - данные импульса
        SIGEV_PULSE_INIT(&event, coid, getprio(0), MY_PULSE_CODE, 0);
        // создаем таймер
        timer_create(CLOCK_REALTIME, &event, &timer_id);

// задаем время срабатывания таймера
        // таймер сработает через 1.8 секунды
        // (1 сек + 800000000 наносек = 1.8 секунды)
        itime.it_value.tv_sec=1;
        itime.it_value.tv_nsec=800000000;

        // таймер повторно сработает через 2.5 секунды
        // (2 сек + 500000000 наносек = 2.5 секунды)
        itime.it_interval.tv_sec=2;
        itime.it_interval.tv_nsec=500000000;

        // создаем относительный таймер (второй параметр равен нулю)
        timer_settime(timer_id, 0, &itime, NULL);

        // теперь мы получим импульс через 1.8 секунды и будем
        // получать повторные через 2.5 секунды

        for(i=10; i>0; i--)
        {
                MsgReceivePulse(chid, &impulse, sizeof(impulse), NULL);
                printf("We got a pulse from our timer\n");
        }
        // получили 10 импульсов и выходим
        return 0;
}
```

Результат выполнения:

```
We got a pulse from our timer
We got a pulse from our timer
We got a pulse from our timer
…
```

Сообщение выводится каждые 2,5 секунды.

Получение значения системного тика

Получить значения системного тика можно с помощью функции clock_getres().

#include <time.h>
*int clock_getres(clockid_t clock_id, struct timespec *res);*

В структуру типа timespec, на которую ссылается указатель res, будет записано разрешение времени часов с идентификатором clock_id. Если в качестве идентификатора clock_id указать константу CLOCK_REALTIME, то получим системный тик.

Таким образом, значение системного тика в наносекундах можно получить с помощью следующего кода:

```
struct timespec res;
clock_getres(CLOCK_REALTIME, &res);
printf("Resolution is %ld nano seconds.\n", res.tv_nsec);
```

Результат выполнения:

```
Resolution is 999847 nano seconds.
```

2.3 Тайм-ауты ядра

Иногда требуется ограничить время проведения потока в блокированном состоянии. Например, клиент, пославший сообщение серверу, будет блокирован до тех пор, пока не получит ответа, но сервер может и не ответить или ответить через неприемлемое для клиента время. В этом случае можно использовать тайм-аут ядра, т.е. ограничение по времени на блокировку потока.

Функция **timer_timeout()** устанавливает тайм-аут на блокировку.

#include <time.h>
*int timer_timeout(clockid_t id, int flags, const struct sigevent *notify, const struct timespec *ntime, struct timespec *otime);*
Здесь

- id – тип временного базиса (для QNX применяется CLOCK_REALTIME);
- flags – задает тип блокировок, для которых ставится тайм-аут;
- notify – указатель на структуру типа sigevent. Перед тем как вызывать функцию структуру нужно инициализировать макросом SIGEV_UNBLOCK_INIT(&notify);
- ntime – время через которое ядро сгенерирует тайм-аут;
- otime – NULL или указатель на структуру timespec, в которую будет записано время проведенное в блокированном состоянии.

Атрибут flags задает тип блокировок, на которые будет установлен тайм-аут. Основные блокировки и функции, которые их вызывают, представлены в таблице 3.

Таблица 3.

Функция, вызывающая блокировку.	Тип блокировки
MsgReceive()	STATE_RECEIVE
MsgSend()	STATE_SEND или STATE_REPLY
pthread_cond_wait()	STATE_CONDVAR
pthread_mutex_lock()	STATE_MUTEX
sem_wait()	STATE_SEM
pthread_join()	STATE_JOIN

Константы для атрибута flag представлены в таблице 4.

Таблица 4.

Тип блокировки	Константа для атрибута flag
STATE_RECEIVE	_NTO_TIMEOUT_RECEIVE
STATE_SEND	_NTO_TIMEOUT_SEND
STATE_REPLY	_NTO_TIMEOUT_REPLY
STATE_CONDVAR	_NTO_TIMEOUT_CONDVAR
STATE_MUTEX	_NTO_TIMEOUT_MUTEX
STATE_SEM	_NTO_TIMEOUT_SEM
STATE_JOIN	_NTO_TIMEOUT_JOIN

Можно задать тайм-аут сразу на несколько блокировок операцией “ИЛИ”, например флаг _NTO_TIMEOUT_SEND|_NTO_TIMEOUT_REPLY приведет к тому, что ядро будет вызывать тайм-аут, когда поток будет переведен в блокировку по передаче (SEND) или по ответу (REPLY).

Примечание: если сервер установил атрибут _NTO_CHF_UNBLOCK при создании канала и принял сообщение, то при срабатывании тайм-аута клиент не разблокируется – ядро лишь пошлет импульс серверу, чтобы он ответил клиенту.

Ниже приведен пример использования тайм-аута ядра. Главный поток запускает дочерний поток, который простаивает 100 секунд. Главный поток устанавливает тайм-аут на блокировку STATE_JOIN и после вызывает функцию **pthread_join(),** которая ждет, пока завершится дочерний поток. Так как был установлен тайм-аут в 5 секунд, то главный поток подождет в блокированном состоянии 5 секунд, и завершится, не дождавшись завершения дочернего потока, который не успеет дойти до оператора return NULL.

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <pthread.h>
#include <errno.h>
#include <time.h>
#include <sys/neutrino.h>

void *the_thread(void *notused)
{       // дочерний поток
  // поток может выполнять работу,
  //   результат которой, если она будет
  //   выполнена не в срок, будет не нужен
  sleep(100); //дочерний поток простаивает 100 секунд
  return NULL;
}

int main(void)
{
  int return_val;
  struct sigevent event;
  struct timespec time;
  pthread_t thread_id;

  // устанавливаем структуру event на разблокирование
  // при срабатывании тайм-аута
  SIGEV_UNBLOCK_INIT(&event);

// устанавливаем время тайм-аута на 5 секунд
  time.tv_sec=5;
  time.tv_nsec=0;

  // создаем поток, который будет выполнять функцию the_thread()
  pthread_create(&thread_id, NULL, the_thread, NULL);

  // устанавливаем тайм-аут на блокировку типа STATE_JOIN
  timer_timeout(CLOCK_REALTIME, _NTO_TIMEOUT_JOIN, &event, &time, NULL);

  // вызываем функцию pthread_join() из-за которой
  //   главный поток блокируется до тех пор, пока не завершится поток thread_id
  return_val=pthread_join(thread_id, NULL);
  if(return_val==ETIMEDOUT)
  {
          // сработал тайм-аут - мы не дождались завершения дочернего потока
          puts("We dont want wait child thread more than 5 seconds!");
  }
  if(return_val==EOK)
  {
          puts("Child thread success terminated.");
  }
  return 0;
}
```

Результат выполнения:

We don’t want wait child thread more than 5 seconds!

Программа выполняется примерно 5 секунд, а не 100 секунд, которые были указаны в дочернем потоке.

3. Порядок выполнения работы

1. Скомпилировать и выполнить приведенные примеры. Проанализировать результаты.
2. Получить значение системного тика в ОС QNX.
3. Разработать и отладить приложение согласно варианту.

Таблица 5

№ варианта	Задание
1	Программа через каждые 5 секунд с помощью таймера выводит время.
2	Программа очищает директорию /tmp каждую минуту. Использовать таймер порождающий поток, который будет выполнять очистку файлов в директории.
3	Аналогично 2-ому варианту, но производить удаление файлов *.core в директории /var/dumps.
4	Программа очищает директорию /tmp каждую минуту. Использовать таймер, посылающий импульс.
5	Аналогично 4-ому варианту, но производить удаление файлов *.core в директории /var/dumps.
6	Программа выдает свободное место в файловой системе / каждую минуту. Использовать таймер, посылающий импульс.
7	Клиент отсылает сообщение серверу. Сервер отвечает некоторым клиентам (функция MsgReply()), а некоторым нет – сделать это с помощью функции rand(). Предусмотреть в клиенте тайм-аут на блокировку. Сделать программу на основе лабораторной работы № 2.
8	Клиент-серверное взаимодействие. Сделать у сервера тайм-аут на блокировку STATE_RECEIVE. Когда серверу не поступают запросы в течение 15 секунд, он выводит сообщение, что нет запросов. Сделать программу на основе лабораторной работы № 2.

Лабораторная работа № 5. Использование среды QNX Momentics IDE.

1. Цель работы

Цель работы - изучение основных принципов использования QNX Momentics IDE.

2. Методические указания

2.1. Общие сведения

Дистрибутив QNX Momentics комплектуется интегрированной средой разработки, которая называется QNX Momentics IDE. За основу для этой IDE разработчики взяли Eclipse, поэтому в Momentics IDE присутствуют многие свойства среды Eclipse. Одной из этих важных свойств является модульная архитектура, которая позволяет разработчикам и потребителям этой технологии свободно модифицировать, встраивать и добавлять возможности, адаптируя возможности среды разработки под свои нужды.

В QNX Momentics входит инструментарий, характерный для многих средств разработки:

- множество систем программирования (C/C++, Java и др.);
- средства визуального моделирования (UML);
- средства разработки графических интерфейсов пользователя;
- средства управления версиями (CVS, SVN).

Также есть более специфичные свойства для построения систем реального времени (в том числе встраиваемых):

- возможность кроссплатформенной разработки;
- комплекты разработчика - разработка драйверов (DDK), средства встраивания (Embedding Tools) и пакет для поддержки различных процессорных плат (BSP).

При открытии Momentics IDE открывается окно, именуемое “рабочим местом” (workbench). "Рабочее место" содержит редакторы и представления. Редактором (editor) называется элемент (plug-in) IDE, позволяющий просматривать и/или модифицировать ресурсы, – например редактор C-файлов, редактор Make-файлов. Представлением (view) называется элемент IDE, позволяющий манипулировать ре-

сурсом, отображая (представляя) его в каком-то логическом виде. К представлениям относятся, например, различные навигаторы и таблицы свойств элементов проектов.

Набор редакторов и представлений, оптимизированный для выполнения какой-либо специализированной задачи, называется перспективой (perspective). Есть ряд предопределенных перспектив:

- C/C++ Development – для проектов на языках C/C++;
- Java – для Java-проектов;
- Debug – для отладки программ;
- Plug-in Development – для разработки новых элементов IDE;
- QNX Application Profiler – для профилирования разработанных программ;
- QNX System Builder – для формирования встраиваемых образов QNX и загрузки их на целевые ЭВМ;
- QNX System Profiler – для визуализации трассы событий ядра, построенной с помощью пакета System Analysis Toolkit;
- QNX System Information – для мониторинга процессоров, выполняющихся на целевых системах.

Сменить текущую перспективу можно выбрав в меню подпункт Window -> Open Perspective -> Other… Количество предопределенных перспектив зависит от установленных дополнений к среде разработки.

По сути, перспектива описывает используемые редакторы и представления, а также их размещение на "рабочем месте". Поэтому можно разместить любые имеющиеся редакторы и представления, так чтоб это было удобно, и объявить полученную конфигурацию рабочего места как новую перспективу (Window -> Save Perspective As…) и потом переключаться на нее .

2.2. Просмотр информации о целевой платформе.

В составе QNX IDE есть эффективное средство мониторинга целевой платформы – перспектива QNX System Information. Чтоб воспользоваться этим средством на целевом узле должна быть запущена программа - агент **qconn**. Чтоб не

запускать **qconn** каждый раз вручную, можно в файл **/etc/rc.d/rc.local** записать следующие строки:

qconn

inetd

Тогда при старте целевой системы будет запускаться агент **qconn** и демон **inetd**, который прослушивает порты и предоставляет различные сервисы.

Для того чтобы просмотреть информацию о целевой платформе, сначала нужно сменить текущую перспективу на QNX System Information. Затем создать проект целевой платформы, выбрав пункт меню File->New->Project… и в группе “QNX” выбрать “QNX Target System Project”. После нажатия кнопки “Next” появится форма, в которой нужно указать имя для проекта, IP-адрес целевой платформы и порт (по умолчанию порт 8000).

В окне Target Navigator появится значок созданного проекта целевой платформы. Если есть какие-нибудь неполадки в соединении, то значок будет с красным крестиком или с восклицательным знаком, и рядом с именем проекта будет написана причина проблемы. Раскрыв список проекта целевой платформы в окне Target Navigator, можно увидеть выполняемые процессы. Чтоб просмотреть информацию о каком-то процессе, нужно выделить его имя в окне Target Navigator и просмотреть вкладки Process Information, Memory Information, Malloc Information. Информация на вкладках описана в таблице 6.

Таблица 6

Вкладка	Информация на вкладке
Process Information	•Сведения о потоках (Thread Details) •Переменные окружения (Environment Variables) •Аргументы (Arguments), переданные процессу, и его идентификаторы (ID)
Memory Information	Используемая процессом память
Malloc Information	Использование кучи, количество вызовов выделения (malloc), освобождения (free), перераспределения (realloc) памяти.

Вкладка System Summary показывает основную информацию о целевой системе:

- имя хоста (Hostname);
- архитектура платформы (Board);

- версия операционной системы (OS Version);
- время загрузки (Boot Date);
- информация о процессоре (CPU Details);
- сведения о памяти (System Memory): занято (Used), свободно (Free), всего памяти (Total);
- количество процессов (Total Process) и информация о них в табличном виде.

Вкладка Target File System Navigator предоставляет файловый менеджер для целевой платформы. С помощью контекстного меню, вызываемого правой кнопкой мыши, можно копировать, удалять, создавать, редактировать файлы.

Таким образом, перспектива QNX System Information предоставляет удобный и наглядный способ получения подробной информации о текущем состоянии целевой платформе.

2.3. Создание проекта

Для создания C/C++ проекта нужно выбрать пункт меню File->New->Project… и в группе “QNX” выбрать “QNX C Project” или “QNX C++ Project” соответственно. После нажатия кнопки Next, на следующей диалоговой форме нужно указать имя проекта (Project name), его расположение (Location) и выбрать тип - программа (application) или библиотека (library). Еще одно нажатие кнопки Next откроет последнее диалоговое окно, содержащее различные опции проекта. Здесь обязательно на вкладке Build Variants необходимо выбрать архитектуру процессора целевой платформы: ARM, PPC, SH или x86.

После нажатия кнопки Finish создастся проект, в котором можно создавать файлы исходных кодов, различных ресурсов и редактировать их во встроенном редакторе. Если текущая перспектива не C/C++, то IDE Momentics всплывающим окошком порекомендует переключиться на эту перспективу.

Свойства проекта можно изменить. Для этого в контекстном меню, вызываемом с помощью нажатия правой кнопкой мыши на имени проекта в окошке Project Explorer, нужно выбрать пункт Properties. Откроется диалоговое окно настройки проекта, где представлено много опций. При нажатии в этом окне на пункт QNX C/C++ Project появится настройки, которые указывались при создании проекта, а также параметры компилятора и компоновщика.

Следует заметить, что библиотека **libc.so**, которая содержит много основных функций, линкуется по умолчанию. Если требуется добавить другие библиотеки, то в окне пункта QNX C/C++ Project следует выбрать вкладку Linker и затем выбрать Extra Libraries в поле Category. Нажать кнопку Add и в столбец Name вписать имя библиотеки без префикса lib и без расширения. К примеру, для добавления стандартной математической библиотеки **libm.so** следует написать m в поле Name.

Для построения проекта в контекстном меню, вызываемого с помощью нажатия правой кнопкой мыши на имени проекта в окошке Project Explorer, нужно выбрать пункт Build Project. В случае наличия ошибок в представлении Problems появится описания ошибок и их местоположение.

2.4. Система контроля версий

В течение разработки сложной программы командой разработчиков просто необходимо отслеживать изменения исходного кода (кто, когда и какова причина изменений). Инструментарий разработки QNX Momentics IDE поддерживает системы контроля версий CVS и SVN. В данной работе будем использовать CVS.

CVS использует архитектуру клиент-сервер. Обычно клиент и сервер соединяются через локальную сеть или через Интернет, но могут работать и на одной машине, если необходимо вести историю версий локального проекта.

Сервер хранит в специальном хранилище (репозитории) текущую версию (версии) проекта и историю изменений, а клиент соединяется с ним, чтобы получить нужную ему версию или записать новую. Получив с сервера нужную версию (данная процедура называется check-out), клиент создаёт локальную копию проекта или его части – так называемую рабочую копию. После того, как в файлы, находящиеся в рабочей копии, внесены необходимые изменения, они пересылаются на сервер (check-in).

Несколько клиентов могут работать над копиями проекта одновременно. Когда они отправляют результаты, сервер пытается слить их изменения в репозитории вместе. Если это не удаётся, например, в случае, когда два клиента изменили одни и те же строки в определённом файле, сервер не примет изменения от последней check-in операции и сообщит клиенту о конфликте, который должен быть исправлен вручную. Если check-in операция завершилась успешно, то номера версий всех

затронутых файлов автоматически увеличиваются, и сервер записывает комментарий, дату и имя пользователя в свой журнал (data logging).

2.4.1 Настройка CVS сервера.

Настроим целевую платформу в качестве CVS сервера:

1) сначала создадим пользователя - владельца репозитория, например anonymous, и войдем в систему под его именем.
2) убедимся, что файл **/etc/services** содержит строчку:

   pserver 2401/tcp

3) создадим папку для хранения репозитория, например, папку cvs в домашней директории пользователя anonymous. Занесем в файл **/etc/inetd.conf** строку (все в одной строке без переносов):

pserver stream tcp nowait root /usr/bin/cvs cvs
-b /usr/local/bin -f --allow-root=root_dir pserver

где вместо root_dir нужно записать путь к папке для хранения репозитория, т.е. /home/anonymous/cvs.

4) теперь выполним команду для инициализации репозитория:

   cvs -d root_dir init

   т.е. команду: cvs -d /home/anonymous/cvs init

   которая создаст служебные файлы в директории /home/anonymous/cvs.

5) в директории root_dir/cvsroot, т.е. в /home/anonymous/cvs/ cvsroot, создадим файл **passwd**, где укажем пользователей репозитория:

   user1::anonymous

   user2::anonymous

   …

Это значит, что пользователям user1, user2, … предоставляется доступ к репозитарию. Причем CVS-пользователи user1, user2 не связаны с пользователями, зарегистрированными в системе, т.к. на самом деле серверная программа CVS выполняется на правах системного пользователя anonymous. Но если user1 внесет изменения в проект, то эти изменения сохранятся под именем user1.

Между двумя двоеточиями ничего не стоит, поэтому используется беспарольный доступ. Пароль можно указать так же, как это делается в файле /etc/passwd.

6) проверим, что в файле **/etc/rc.d/rc.local** есть строка для старта демона **inetd** при запуске операционной системы:

inetd

Теперь после перезагрузки целевой системы или перезапуска демона inetd (команда slay inetd; inetd из-под root-пользователя), мы получим настроенный CVS-сервер на целевой машине.

2.4.2 Использование CVS через IDE (клиент CVS)

Чтоб использовать репозиторий на клиентском хосте, сначала нужно указать его местоположение. На клиентском хосте в среде разработки QNX Momentics IDE переключимся в перспективу CVS Repository Exploring (Window->Open Perspective->Other..). В представлении CVS Repositories отображается список репозиториев, в который необходимо добавить репозиторий, созданный на целевой системе.

Из контекстного меню, вызываемого нажатием правой кнопки мыши в области представления CVS Repositories, выберем пункт New->Repository Location… В диалоговом окне в поле Host указываем IP-адрес CVS-сервера, в поле Repository path - путь к репозиторию (/home/anonymous/cvs), User – пользователь CVS (user1), в поле Password – пароль (пустой), в поле Connection Type выбираем pserver (тип соединения характерный для CVS). После нажатия кнопки Finish репозиторий станет доступен для среды разработки.

CVS имеет много возможностей, рассмотрим основные:

1) передача проекта под управление CVS.

Переключимся на перспективу C/C++ (Window->Open Perspective->Other..). Выберем нужный проект, щелкнем по нему правой кнопкой и выберем пункт Team->Share Project… В диалоговом окне выберем тип репозитория – CVS, в следующем окне выберем репозиторий, чье местоположение было указано ранее. В следующем окне нужно указать имя модуля. Модуль с точки зрения CVS — это отдельный блок информации, который весь целиком может быть запрошен пользователем. Обычно для каждого проекта или группы проектов заводится свой CVS-модуль, поэтому

можно оставить именем модуля имя проекта (Use project name as module name). В следующем диалоговом окне отобразятся файлы проекта, которые передаются под управления CVS. После нажатия кнопки Finish откроется диалоговое окно Commit Files, в котором следует ввести комментарий внесенных изменений, например “Это начальная версия проекта”. Теперь проект под управлением CVS.

2) внесение изменений в репозиторий.

Когда разработчик изменяет и сохраняет файл, то это автоматически записывается в локальную историю. А для того, чтобы отправить изменения в репозиторий на CVS-сервере, нужно из контекстного меню проекта в представлении Project Explorer, выбрать пункт Team->Commit… В появившемся диалоговом окне нужно записать комментарий к изменениям, также здесь в области changes отобразятся файлы, которые были изменены. После нажатия кнопки Finish изменения отправится в репозиторий на CVS-сервере (check-in процедура) и, если не возникнет конфликтов с изменениями других разработчиков, операция произойдет успешно.

После внесения изменения версия файла автоматически изменится, к примеру с 1.1. на 1.2.

3) просмотр истории изменений.

Для просмотра изменений файла, из контекстного меню нужного файла в представлении Project Explorer следует выбрать пункт Team->Show history. На вкладке History отобразится история изменений в виде таблицы со следующими столбцами: Revision (номер версии), Revision Time (время создания версии), Author (кто внес изменения), Comment (коментарий).

Рядом есть кнопки для выбора отображения локальных (local) и/или на CVS-сервере (remote) версий: Local Revisions, Remote Revisions, Local Revisions and Remote Revisions. Версии можно открыть двойным щелчком мыши. Если нажать правой кнопкой по какой-нибудь из версии на вкладке History и выбрать пункт Compare current with… то можно получить наглядное сравнение текущий версии и выбранной.

2.5. Запуск проекта на целевой платформе

Для запуска или отладки построенного проекта на целевой платформе нужно создать конфигурацию запуска (Launch Configuration). Она состоит из различных

настроек, указывающих как запускать программу (входные параметры, переменные окружения). Эти настройки задаются один раз, а затем их можно использовать много раз для запуска программы.

Чтоб создать конфигурацию для отладки, нужно на панели инструментов выбрать пункт Run ->Open Debug Dialog.… Откроется диалоговое окно, слева выберите C/C++ QNX QConn (IP). Теперь нажмите на иконку New launch configuration.

Появится много вкладок, основные настройки находятся на вкладке Main. В поле Project с помощью кнопки Browse необходимо указать проект, который требуется запустить для отладки, Далее заполнить поле C/C++ Application, нажав кнопку Search Project… и выбрав бинарный файл для запуска. Для отладки используются бинарные файлы с суффиксом _g , т.к эти файлы содержат в себе отладочную информацию. Следует убедиться, что в Target Options должна быть указана конфигурация целевой платформы и после нажатия кнопки Apply конфигурация будет создана.

Теперь, когда конфигурация создана, в диалоговом окне можно нажать кнопку Debug. Momentics IDE переключится на перспективу Debug, программа запустится и остановится на первой строке кода. На панели представления Debug находятся элементы управления отладкой, аналогичные применяемым в других средах разработок. Так нажатие кнопки Step Over переведет исполнение на следующую строку кода в пределах текущей функции, а нажатие кнопки Step Into переведет исполнение внутрь вызываемой функции. Двойной щелчок по закрашенной области слева от исполняемого кода установит точку останова (Breakpoint). Значения переменных отображаются в представлении Variables. Для прекращения отладки необходимо нажать кнопку Terminate и, если нужно, сменить перспективу (например, на C/C++).

Для простого запуска приложения (без отладки) требуется выбрать на панели инструментов пункт Run->Open Run Dialog.… Настройки здесь аналогичны настройкам, применяемым для конфигурации отладки. Для запуска можно использовать и бинарный файл с отладочной информацией (суффикс _g), только эта информация не будет использована, поэтому возможно более медленное исполнение по сравнению с использованием файла без отладочной информации (без суффикса _g).

При последующих запусках через Open Run Dialog/Open Debug Dialog слева будут отображаться уже созданные конфигурации и из них можно просто выбрать нужную.

2.6. Профилирование

Под профилированием в системах реального времени называют измерение производительности всей программы или отдельных ее фрагментов с целью нахождения “горячих” точек (Hot Spots) - участков программы, на выполнение которых расходуется наибольшее количество времени. Оптимизация этих точек позволяет наиболее увеличить производительность программы.

В QNX Momentics IDE профилирование выполняется с помощью перспективы QNX Application Profiler. Поддерживаются следующие типы профилирования:

- Sampling – способ сбора статистики, который не требует специальной компиляции проекта. Сбор информации происходит через постоянные интервалы времени. Метод имеет низкие накладные расходы, но программа должна выполняться достаточно долго для получения корректных данных профилирования;
- Function Instrumentation profiling – способ, который представляет довольно точную информацию о времени исполнения функций проекта. Компилятор вставляет инструкции замера в начале и в конце функций проекта. Этот метод лучше работает на однопоточных программах, потому что в случае многопоточности накладные расходы измерений могут изменить поведение программы.
- Sampling and Call Count instrumentation profiling – этот метод является расширением первого метода путем добавления информации о количествах вызовов функций. Компилятор включает перед каждой функцией код для сбора данных о количествах вызовов. Если существенное время в программе занимает выполнение библиотечных подпрограмм, то вероятно снижение значений результатов профилирования, в этом случае нужно использовать Function Instrumentation profiling.

Для профилирования вторым или третьим способом (Function Instrumentation profiling или Sampling and Call Count instrumentation profiling) требуется специальная компиляция проекта. Для этого в свойствах проекта (Project->Properties) слева нужно выбрать пункт “QNX C/C++ project” и на вкладке Options выбрать “Build for Profiling (Function Instrumentation)” или “Build for Profiling (Call Count Instrumentation)”. После перестроения проекта в исполняемом файле будут содержаться инструкции для профилирования.

Для запуска профилирования сначала удобнее сразу переключится в перспективу QNX Application Profiler (Window -> Open Perspective -> Other…). Далее нужно в диалоговом окне, вызываемым командой Run -> Profile…, создать конфигурацию так же как это делается для обычного запуска приложения или отладки. Для более информативных результатов профилирования лучше выбрать файл с отладочной информацией (с суффиксом _g), это позволит среде разработке связать исполняемый код со строками исходного кода. Здесь следует учесть, что для режима Function Instrumentation не рекомендуется применять файл с отладочной информацией, так как это может исказить результаты профилирования.

Когда настройки на вкладке Main будут указаны, нужно переключиться на вкладку “Tools”, где нажать кнопку “Add/Delete Tool…” и выбрать инструмент “Application Profiler”. Теперь в области Profiling Method нужно выбрать метод профилировки: Function Instrumentation или Sampling and Call Count instrumentation profiling. Если выбрана опция Sampling and Call Count instrumentation profiling и в настройках проекта не указана компиляция для профилировки, то будет использоваться только метод Sampling (без подсчета количества вызовов и изменений кода приложения).

После запуска приложения среда разработки начнет собирать информацию по выполнению. Когда приложение завершится, в представлении Profiler Sessions отобразится сессия профилировки. Чтоб просмотреть профилировочную информацию необходимо двойным щелчком открыть нужную сессию, при этом сведения, которые зависят от типа профилировки, откроются в представлении Execution Time.

2.7. Примеры приложений

Вариант 1.

Рассмотрим пример фундаментального алгоритма – сортировка выбором. Этот алгоритм сортировки заключается в следующем. Сначала отыскивается наименьший элемент массива, затем он меняется местами с элементом, стоящим первым в сортируемом массиве. Далее находится наименьший элемент из оставшейся части массива и меняется с элементом стоящим вторым в исходном массиве. Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован. Т.е. принцип заключается в выборе наименьшего элемента из числа неотсортированных.

Программа сортирует строки по алфавиту, хотя на самом деле сортируются указатели на строки, так как нет смысла в перемещении строк в памяти, это изображено на рисунке 5.

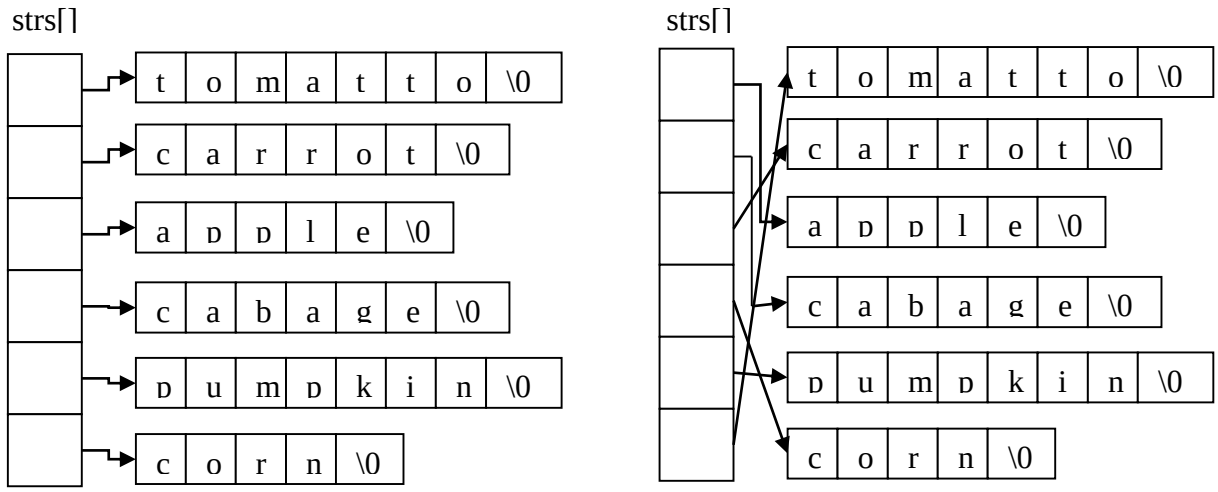


Рис 5. Порядок указателей до и после сортировки.

Для сравнения строк используется стандартная функция **strcmp(char* str1, char* str2),** которая возвращает отрицательное значение, если строка str1 меньше строки str2.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *strs[]={"tomatto", "carrot", "apple",
                "cabage", "pumpkin", "corn"};

void printstrs(char **mas, int n)
{
        int j;
        for(i=0; i<n; i++)
        {
                printf("\t%s\n", mas[i]);
```

```
        }
}

void sorting_str(char **mas, int n)
{
        int i, j, min;
        char *temp;
        for(i=0; i<n-1; i++)
        {
                min=i;
                for(j=0; j<n; j++)
                {
                        if(strcmp(mas[j], mas[min])<0)
                                min=j;
                }
                temp=mas[i];
                mas[i]=mas[min];
                mas[min]=temp;
        }
}

int main(int argc, char *argv[])
{
        int n;
        n=sizeof(strs)/sizeof(char*);
        puts("Original strings:");
        printstrs(strs, n);
        sorting_str(strs, n);
        puts("Sorted strings:");
        printstrs(strs, n);
        return EXIT_SUCCESS;
}
```

Вариант 2.

Сортировка вставками заключается в том, что берется элемент из неотсортированного участка и помещается в нужное место среди уже отсортированных элементов.

Анализируемый элемент продвигается по уже отсортированной части массива, сдвигая элементы влево, пока не встанет на нужную, но возможно не окончательную для него позицию.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *strs[]={"tomatto", "carrot", "apple", "cabage",
                        "pineapple", "pumpkin", "corn"};

void printstrs(char **mas, int n)
```

```
{
	int i;
	for(i=0; i<n; i++)
	{
		printf("\t%s\n", mas[i]);
	}
}

void sorting_str(character **mas, int n)
{
	int i, j;
	char *temp;

	for(i=0; i<n; i++)
	{
		for(j=i; j>1; j--)
		{
			if(strcmp(mas[j-1], mas[j])>0)
			{
				temp=mas[j-1];
				mas[j-1]=mas[j];
				mas[j]=temp;
			}
		}
	}
}

int main(int argc, char *argv[])
{
	int n;

	n=sizeof(strs)/sizeof(char*);
	puts("Original strings:");
	printstrs(strs, n);

	sorting_str(strs, n);

	putstr("Sorted strings:");
	printstrs(strs, n);
	return EXIT_SUCCESS;
}
```

Вариант 3.

Еще один фундаментальный алгоритм сортировки – это пузырьковая сортировка. Алгоритм состоит в многократном проходе массива с обменом местами соседних элементов, нарушающих заданный порядок, до тех пор пока массив не будет отсортирован.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *strs[]={"tomatto", "carrot", "apple", "cabage",
		"pineapple", "pumpkin", "corn"};
```

```
void printstrs(char **mas, int n)
{
	int i;
	for(i=0; i<n; i++)
	{
		printf("\t%s\n", mas[i]);
	}
}

void sorting_str(char **mas, int n)
{
	int i, j;
	char *temp;

	for(i=0; i<n-1; i--)
	{
		for(j=n-1; j>i; j--)
		{
			if(strcmp(mas[j-1], mas[j])>0)
			{
				tmp=mas[j-1];
				mas[j-1]=mas[j];
				mas[j]=temp;
			}
		}
	}
}

int main(int argc, char *argv[])
{
	int n;

	n=sizeof(strs)/sizeof(char*);
	puts("Original strings:");
	printstrs(strs, n);

	sorting_str(strs, n);

	puts("Sorted strings:");
	printstrs(strs, n);
	return EXIT_SUCCESS;
}
```

3. Порядок выполнения работы

1. С помощью средств перспективы QNX System Information определить следующую информацию о целевой платформе:

- архитектуру и частоту процессора;
- общий объем оперативной памяти и объем занятой памяти;
- версию операционной системы QNX;

- количество запущенных процессов;
- количество потоков у процесса **pipe** и их состояние.

2. Создать QNX C Project с кодом приложения согласно варианту. Передать проект под управление CVS.

3. Скомпилировать проект, исправить синтаксические ошибки, если такие имеются. Занести исправления в репозиторий с описанием изменений.

4. Создать конфигурацию для запуска отладки и выполнить пошаговую отладку приложения. Внесенные исправления также заносить в репозиторий с описанием.

5. Создать конфигурацию для обычного запуска без отладочной информации и выполнить запуск приложения. Сравнить размеры исполняемых файлов с отладочной информацией и без нее (пункт Properties в контекстном меню нужного файла в представлении Project Explorer).

6. Выполнить профилирование методами “Function Instrumentation profiling” и “Sampling and Call Count instrumentation profiling”. Из результатов получить время выполнения функций программы для обоих методов. Занести данные по времени в таблицу:

Функция	Затраченное время
main	
sorting_str	
print_str	

Лабораторная работа № 6 Алгоритмы планирования СРВ

1. Цель работы

Целью работы является изучение алгоритмов планировщика задач для систем реального времени.

2. Методические указания

Целью планирования в системах реального времени (СРВ) является выполнение процессов в течение заранее определенного интервала времени. Алгоритм планирования должен определять последовательность выполнения задач в соответствии с их требованиями к ресурсам и времени исполнения. Выбор алгоритма планирования при проектировании СРВ может зависеть от ряда факторов: количества задач и процессоров в системе, типа задач, отношения предшествования между задачами, метода синхронизации задач и т.д.

В СРВ используются в основном вытесняющие алгоритмы планирования, основанные на возможности прерывания выполнения активной задачи в случае поступления запроса от задачи с более высоким приоритетом. Такие алгоритмы могут быть статическими и динамическими.

Статические алгоритмы определяют план выполнения задач по их априорным характеристикам, назначая каждой задаче фиксированное значение приоритета на все время ее нахождения в системе. Они характеризуются низкими затратами ресурсов, но требуют полной предсказуемости СРВ.

Динамические алгоритмы определяют текущий план во время исполнения задач и имеют возможность изменения их приоритета. Динамическое планирование связано со значительными затратами, но способно адаптироваться к меняющемуся окружению.

Примером статического вытесняющего алгоритма является алгоритм монотонных частот RMS (Rate Monotonic Scheduling), динамического – алгоритм EDF (Earliest Deadline First), отдающий предпочтение задачам с наиболее ранним предельным временем начала или завершения выполнения.

2.1. Алгоритм монотонных частот

2.1.1 Описание алгоритма

Алгоритм монотонных частот (RMS) используется для периодических и апериодических задач. Напомним, что основными характеристиками периодических задач являются: период запуска T; максимальное время выполнения C (время процессора, необходимое для завершения одного запуска), в некоторых источниках обозначается как P; максимальная задержка передачи сигналов управления и реализации управляющего воздействия D; приоритет, значение которого устанавливается обратно пропорциональным периоду запуска. Вводится также понятие коэффициента использования процессора, под которым при допущении о малости задержек понимается отношение $U = C/T$.

Алгоритм RMS назначает приоритет задачи обратно пропорционально ее периоду, т.е. чем меньше период задачи, тем выше ее приоритет. Задачи должны удовлетворять следующим условиям:

- задача должна быть завершена за время ее периода;
- задаче требуется одинаковое процессорное время на каждом периоде;
- задачи являются независимыми;
- прерывание задачи происходит мгновенно;
- приоритеты у всех задач должны быть фиксированными и иметь различные значения;
- апериодические задачи не имеют жестких сроков.

2.1.2 Теория планирования в реальном времени

На этапе проектирования СРВ важным этапом является определение производительности системы. Если система не справляется со своими задачами в течение отведенного интервала, последствия могут быть весьма серьезными. Анализ производительности проекта основан на теории планирования в реальном времени. Этот метод предназначен для СРВ, которые должны выдерживать жесткие временные ограничения и позволяет определить, будут ли выполнены наложенные ограничения.

В *теории планирования в реальном времени* рассматриваются вопросы приоритетного планирования параллельных задач с жесткими временными ограничениями. Эта теория позволяет предсказать, будет ли группа задач, для каждой из которых потребление ресурсов процессора известно, удовлетворять этим ограничениям при использовании статического алгоритма RMS. Изначально алгоритмы планирования в реальном времени разрабатывались для независимых периодических задач, то есть таких периодических задач, которые не взаимодействуют друг с другом и, следовательно, не нуждаются в синхронизации.

Задача называется *планируемой*, если она удовлетворяет всем временным ограничениям, то есть ее исполнение завершается до истечения периода. Группа задач будет *планируемой***,** когда планируемой является каждая входящая в нее задача.

В теории планирования доказана следующая теорема о верхней границе коэффициента использования процессора: *множество из n независимых периодических задач, планируемых согласно алгоритму RMS, всегда удовлетворяет временным ограничениям, если*

$$C_1/T_1 + \ldots + C_n/T_n <= n\,(2^{1/n} - 1) = U(n),$$

где C_i и T_i – время выполнения и период i – ой задачи соответственно.

Значения верхней границы U(n) для числа задач от 1 до 9 приведены в табл. 7. Это оценка для худшего случая (когда все задачи готовы к выполнению одновременно), а для случайно выбранной группы задач верхняя граница равна 88%. Если периоды задач гармоничны (являются кратными друг другу), то верхняя граница оказывается еще выше.

Таблица 7

Число задач n	1	2	3	4	5	6	7	8	9	∞
Верхняя граница U(n)	1,000	0,828	0,779	0,756	0,743	0,734	0,728	0,724	0,72	0,69

Достоинство алгоритма монотонных частот заключается в том, что он сохраняет устойчивость в условиях краткосрочной перегрузки. Подмножество всего множества задач, состоящее из задач с наивысшими приоритетами (то есть наименьшими периодами), все еще будет удовлетворять временным ограничениям, если система в течение короткого промежутка времени подвергается сверхрасчетной нагрузке. Задачи с низкими приоритетами по мере повышения загрузки процессора могут эпизодически выполняться дольше положенного времени.

Применим теорему о верхней границе коэффициента использования к трем задачам со следующими характеристиками (время выражено в мсек):

Задача z_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0,2$.

Задача z_2: $C_2 = 30$; $T_2 = 150$; $U_1 = 0,2$.

Задача z_3: $C_3 = 60$; $T_3 = 200$; $U_3 = 0,3$.

Здесь предполагается, что накладные расходы на контекстное переключение, требуемые в начале и в конце выполнения задачи, содержатся в оценке времени процессора.

Полный коэффициент использования процессора для всех трех задач равен $0,7<0,779$, поэтому эти задачи будут всегда удовлетворять временным ограничениям. При увеличении времени выполнения третьей задачи до 90 мсек ($U_3=0,45$) полный коэффициент использования процессора равен $0,85 > 0,779$, следовательно теорема о верхней границе не дает гарантии, что задачи смогут удовлетворить временным ограничениям. Поэтому необходимо получить более точную оценку с помощью теоремы о времени завершения.

2.1.3 Теорема о времени завершения

Теорема о времени завершения дает более достоверный критерий для оценки производительности СРВ и используется, если для некоторого множества задач полный коэффициент использования процессора больше, чем требует теорема о верхней границе.

Теорема о времени завершения: *если имеется такое множество независимых периодических задач, в котором каждая задача успевает завершиться вовремя в случае, когда все задачи запускаются одновременно, то все задачи смогут завершиться вовремя при любой комбинации моментов запуска.*

В этой теореме предполагается худший случай, когда все периодические задачи готовы к исполнению одновременно. Она позволяет точно определить, является ли данное множество независимых периодических задач планируемым. Если в указанных условиях выполнение задачи завершается до истечения ее первого периода, то она всегда будет удовлетворять временным ограничениям.

Чтобы убедиться в выполнении условий теоремы, необходимо проверить момент завершения первого периода для данной задачи z_i, а также моменты завершения периодов всех задач с более высоким приоритетом. Согласно алгоритму RMS периоды подобных задач будут меньше, чем для задачи z_i. Эти периоды называются точками планирования. Задача z_i один раз займет процессор на время C_i в течение своего периода T_i, но более приоритетные задачи будут выполняться чаще и могут по крайней мере один раз вытеснить задачу z_i.

Теорема о времени завершения графически представляется с помощью временной диаграммы, на которой показана упорядоченная последовательность выполнения группы задач. Рассмотрим выполнение группы из трех задач со следующими характеристиками в однопроцессорной системе:

задача z_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0,2$;

задача z_2: $C_2 = 30$; $T_2. = 150$; $U_2 = 0,2$;

задача z_3: $C_3 = 90$; $T_3 = 200$; $U_3 = 0,45$;

суммарный коэффициент использования процессора – 0,85.

Их выполнение показано на временной диаграмме (рис. 1), темные участки обозначают интервалы выполнения задач. В худшем случае, когда все три задачи готовы к работе одновременно, первой запускается z_1 , так как у нее самый короткий период и самый высокий приоритет. Она завершается через 20 мс, после чего в течение 30 мс исполняется задача z_2, а затем z_3. В конце первой точки планирования T_1=100 мс задача z_1 уже завершена и, следовательно, удовлетворяет временным ограничениям. Задача z_2 также завершила работу задолго до срока, а задача z_3 потратила 50 мс из необходимых 90.

В начале второго периода z_1 задача z_3 вытесняется задачей z_1 , которая через 20 мс завершается и уступает процессор задаче z_3. Задача z_3 выполняется до конца периода T_2 (150 мс), который соответствует второй точке планирования. В этот момент z_3 использовала 80 мсек из необходимых 90.

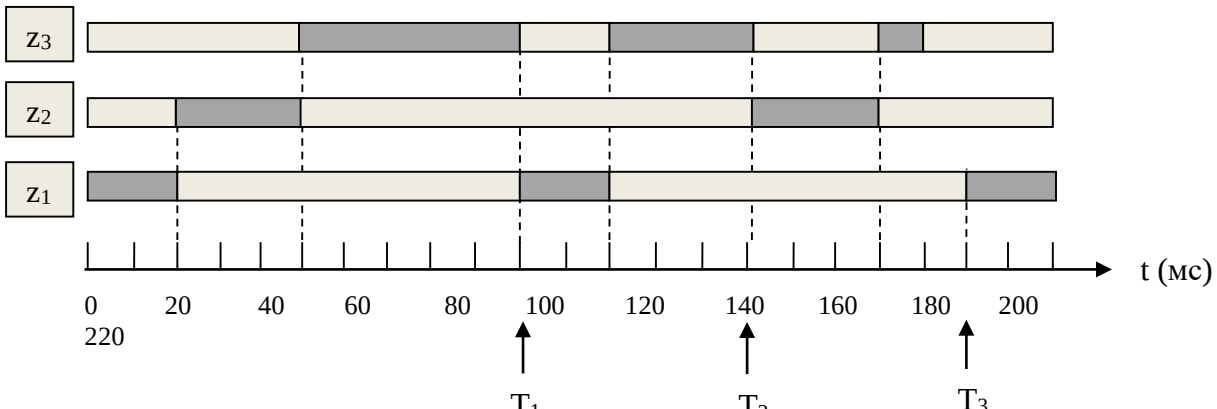


Рис. 1. Временная диаграмма алгоритма RMS

В момент времени T_2 задача z_2 вытесняет задачу z_3 и через 30 мс возвращает процессор задаче z_3, которая исполняется еще 10 мс, укладываясь в отведенное для нее время. Третья точка планирования T_3, которая расположена в конце второго периода задачи z_1 ($2T_1 = 200$) и одновременно в конце первого периода задачи z_3 ($T_3 = 200$). Из рисунка видно, что все задачи завершают исполнение до конца своего первого периода, поэтому все вместе они удовлетворяют временным ограничениям.

На диаграмме видно, что процессор простаивает 10 мс перед началом третьего периода z_1 (этот момент совпадает с началом второго периода z_3). На протяжении интервала длительностью 200 мс процессор работал 190 мс, т.е. задействовался на 95%. По истечении времени, равного наименьшему общему кратному трех периодов (600 мс), средний коэффициент использования окажется равным 0,85.

2.2. Алгоритм EDF

Динамический алгоритм EDF устанавливает максимальный приоритет задачам, у которых раньше всех требуется очередной запуск, т.е. раньше крайний срок исполнения.

Имеются два варианта алгоритма EDF - без вытеснения и с вытеснением задач. При невытесняющем EDF активная задача всегда доделывает свою работу до конца независимо от того, появились ли во время работы этой задачи другие задачи с более высоким приоритетом. При использовании режима вытеснения активная задача принудительно прерывается задачей, имеющей более высокий приоритет и перешедшей в состояние готовности.

Планировщик EDF ведет отсортированную по крайним срокам очередь готовых к работе задач. Каждая задача, перешедшая в состояние готовности, регистрируется в этой очереди и объявляет о своем крайнем сроке. Согласно алгоритму EDF всегда запускается первая задача из очереди, то есть та, у которой раньше всех наступает крайний срок. Запуск алгоритма EDF (точка планирования) возникает

каждый раз при завершении очередной задачи или переходе какой – либо задачи в состояние готовности.

Если используется алгоритм EDF с вытеснением, то при переходе в состояние готовности новой задачи планировщик проверяет, не наступает ли ее крайний срок раньше, чем у активной задачи и при положительном ответе активная задача вытесняется новой задачей.

Основные свойства алгоритма EDF:

- каждая задача должна быть завершена за время ее периода;
- задачи являются независимыми;
- прерывание задач происходит мгновенно;
- может работать с периодическими и апериодическими задачами.
- не требует постоянства времени использования процессора задачами;

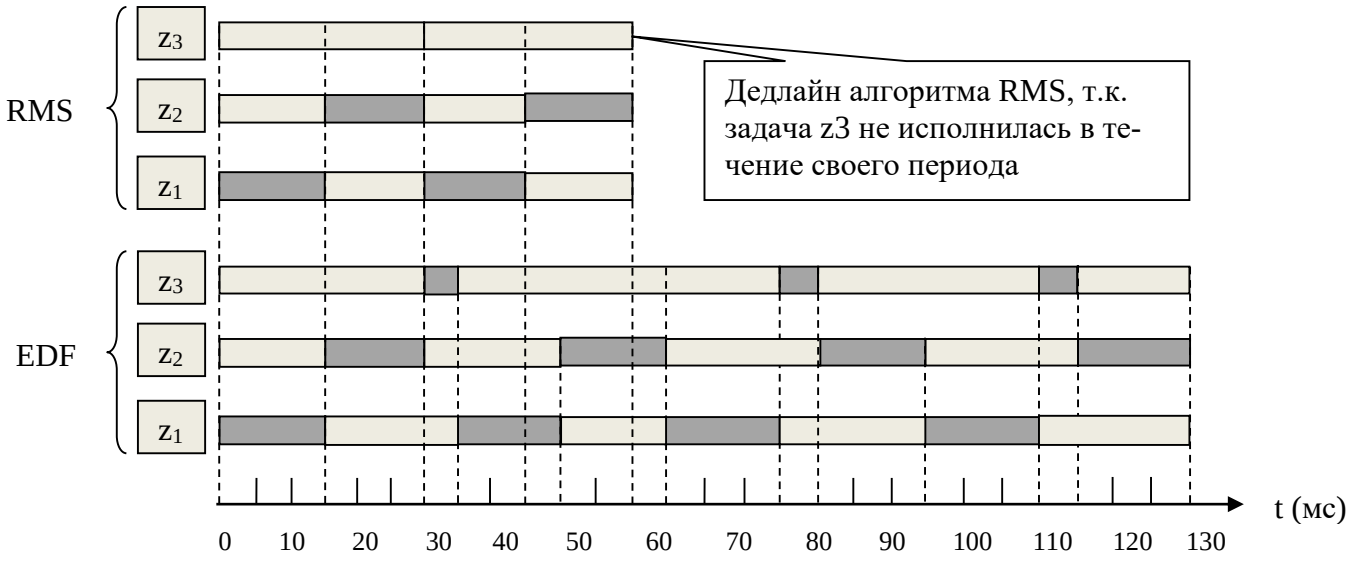


Рис. 2 – Временная диаграмма алгоритмов EDF и RMS

Рассмотрим работу не вытесняющего алгоритма на примере выполнения следующей группы задач в однопроцессорной системе:

задача z_1: $C_1 = 15$; $T_1 = 30$; $U_1 = 0,5$;

задача z_2: $C_2 = 15$; $T_2. = 40$; $U_2 = 0,375$;

задача z_3: $C_3 = 5$; $T_3 = 50$; $U_3 = 0,1$;

суммарный коэффициент использования процессора – 0,975.

В таблице 8 приведена схема работы алгоритма EDF в каждой точке планирования на интервале до 120 мсек, а на рис. 2 приведена временная диаграмма выпол-

нения этой группы задач. Для сравнения там же показана работа статического алгоритма RMS, который в момент времени t=60 мсек дает сбой из-за того, что задача z3 не может выполниться в течение своего первого периода.

Таблица 8

Точка планирования, мсек	Крайний срок завершения задачи, мсек			Выбор для исполнения
	z1	z2	z3	
0	30	40	50	z1-1
15	Не готова	40	50	z2-1
30	60	Не готова	50	z3-1
35	60	Не готова	Не готова	z1-2
50	Не готова	80	100	z2-2
65	90	Не готова	100	z1-3
80	Не готова	120	100	z3-2
85	Не готова	120	Не готова	z2-3
100	120	Не готова	150	z1-4
115	Не готова	Не готова	150	z3-3

2.3 Программное обеспечение

Для выполнения лабораторной работы предлагается использовать специальное программное обеспечение, разработанное в НГТУ – эмуляторы планировщиков реального времени.

Приложение MiltiPlanner позволяет продемонстрировать работу алгоритмов RMS и EDF. Благодаря модульной структуре программа в любой момент может быть дописана и включить в себя ещё множество интересных и показательных алгоритмов. На рис. 1 представлен интерфейс приложения.

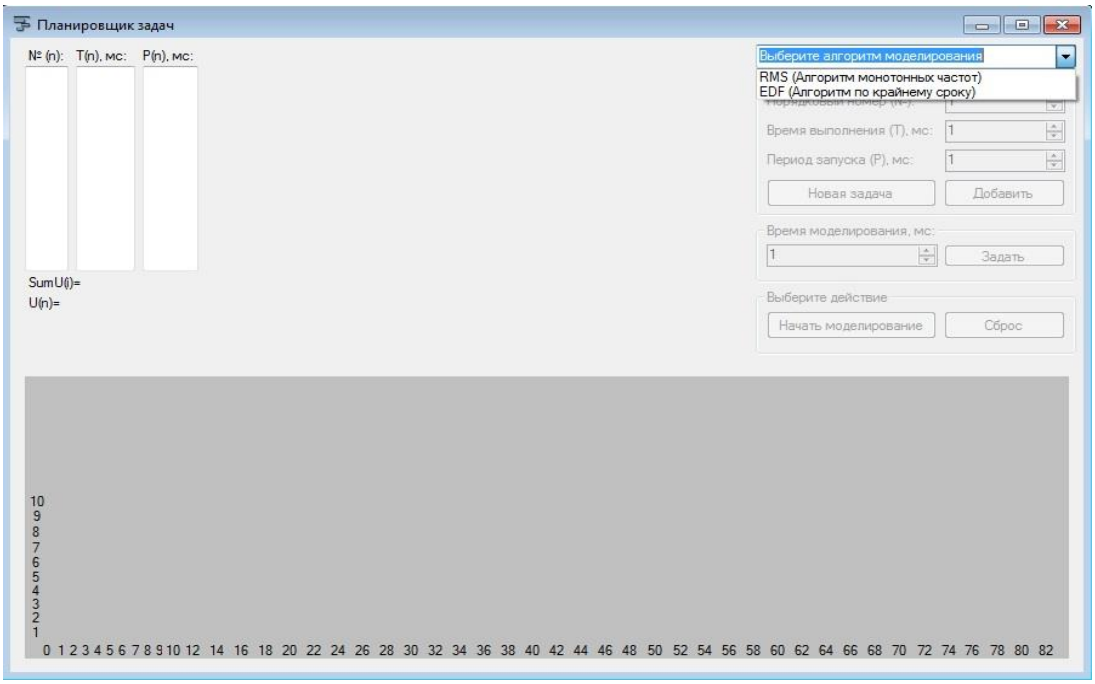


Рис. 1. Интерфейс эмулятора MultiPlaner

Блок ввода исходных данных располагается в правой верхней части. Он содержит поле выбора алгоритма, три поля для ввода параметров задач (порядковый номер, время выполнения и период запуска) и поле ввода времени моделирования. Ввод начинается с выбора алгоритма, после чего становятся доступными все остальные поля. Для примера введем следующий набор задач:

- задача № 1, время выполнения T = 2 мс, период запуска P = 10 мс.
- задача № 2, время выполнения T = 3 мс, период запуска P = 15 мс.
- задача № 3, время выполнения T = 9 мс, период запуска P = 20 мс.

После нажатия кнопки «Добавить» параметры задачи переносятся в таблицу задач и можно продолжить ввод очередной задачи нажатием на кнопку «Новая задача». По окончании ввода задач необходимо указать время моделирования и нажать кнопку «Начать моделирование».

Результаты работы планировщика по указанной группе задач выводятся в виде временной диаграммы, отображающая график распределения процессорного времени по задачам. Горизонтальная ось соответствует времени, вертикальная ось соответствует номеру задачи.

Зелёным цветом представлены задачи и то, в каком порядке они были отправлены планировщиком на обработку процессору. Чёрные блоки – это периоды простоя процессора, когда ни одна задача не находилась в состоянии готовности.

Также для выбранной группы задач выводятся значения фактического и расчетного коэффициентов загрузки процессора (рис. 2).

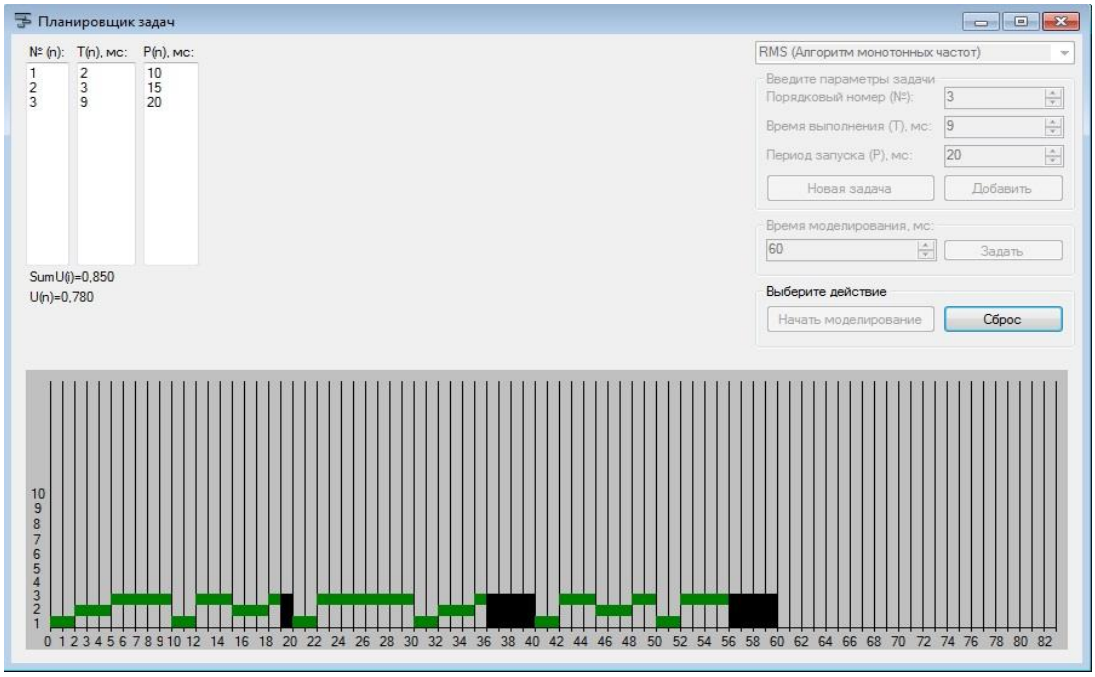


Рис. 2. Результаты моделирования

Для анализа нового набора задач или изменения алгоритма моделирования необходимо нажать на кнопку «Сброс».

Алгоритм эмулятора предполагает остановку в том случае, когда какая-либо из задач не успевает завершиться за время своего периода (рис. 3). Такая ситуация называется дедлок (Deadlock).

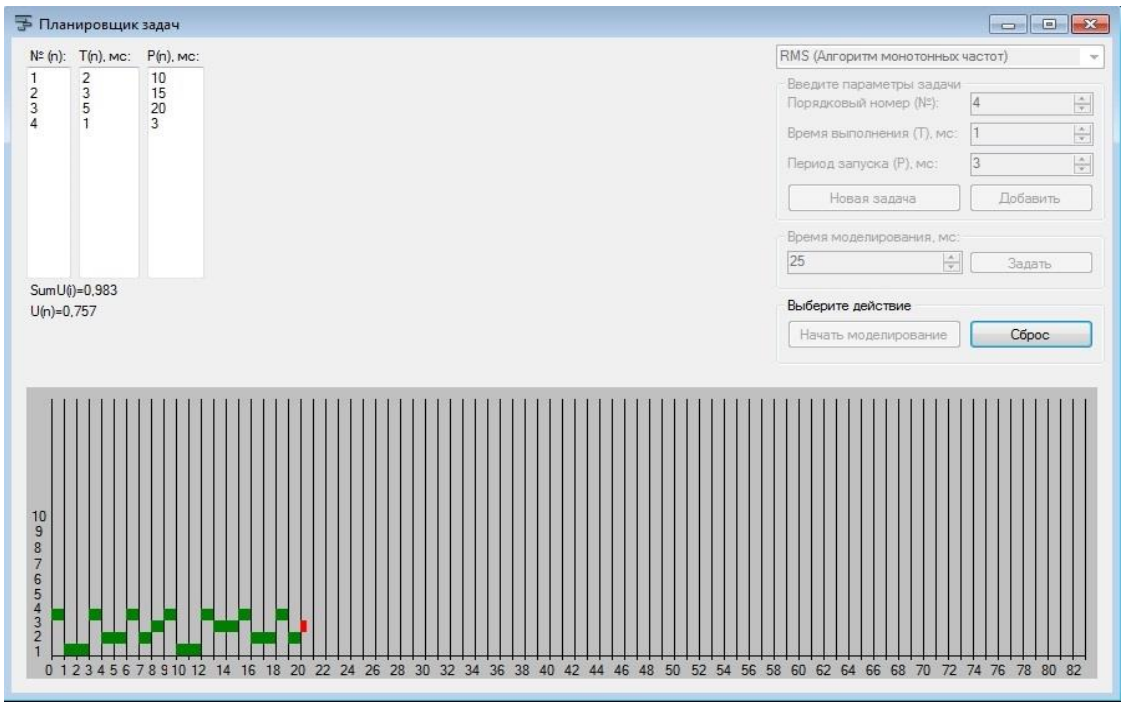


Рис. 3. Дедлок задачи № 3. Моделирование остановлено

На рис. 4 показан пример успешного выполнения группы задач с расчетной загрузкой процессора 98,3 % с помощью динамического алгоритма EDF. Параметры задач:

- задача № 1, время выполнения T = 2 мс., период запуска P = 10 мс;
- задача № 2, время выполнения T = 3 мс., период запуска P = 15 мс;
- задача № 3, время выполнения T = 5 мс., период запуска P = 20 мс;
- задача № 4, время выполнения T = 1 мс., период запуска P = 3 мс.

На временной диаграмме видно, что процессор за 60 мс сделал 59 рабочих вычислительных тактов, что полностью совпадает с расчетной загрузкой: 59/60 = 0,983.

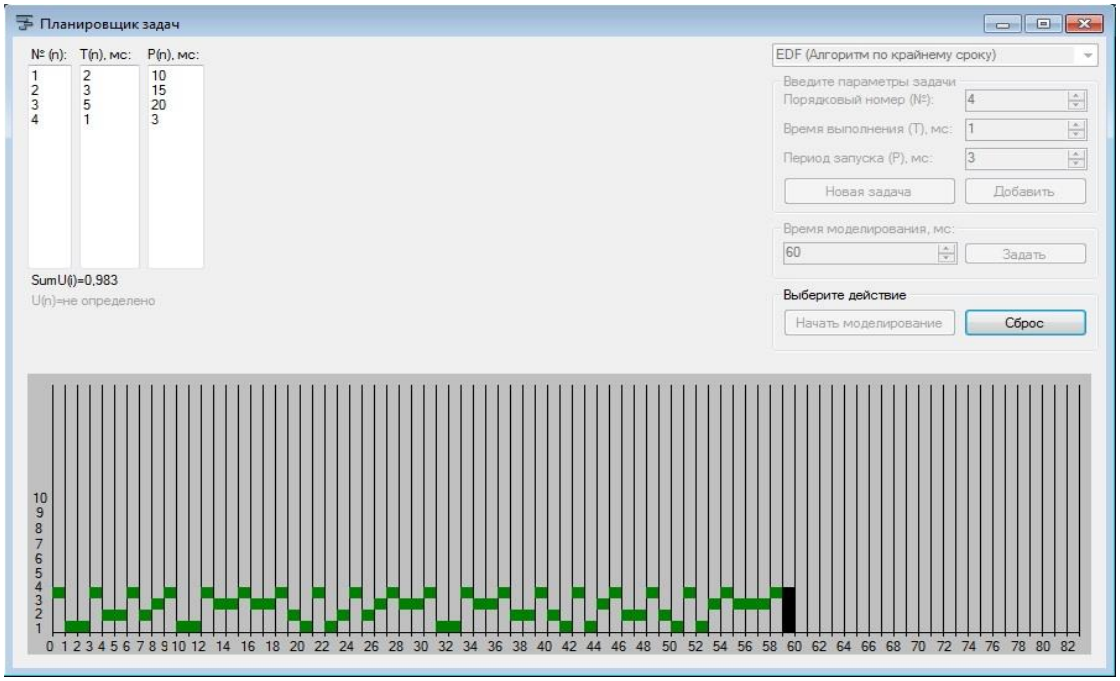


Рис. 4. Моделирование алгоритма EDF

Напомним, что для статического алгоритма RMS согласно таблицы 7 верхняя граница загрузки процессора для четырех задач составляет 75,6%.

Приложение MiltiPlanner_v2 является усовершенствованной версией эмулятора MiltiPlanner и отличается от базовой версии следующими особенностями:

- возможность эмуляции алгоритмов RMS, EDF и LLF;
- более совершенные алгоритмы эмуляции, позволяющие существенно увеличить время моделирования.

Меню программы включает два пункта:

- *Задачи* – позволяет управлять списком задач: загрузить список из файла или очистить текущий список;

- *Справка* – открывает справочные материалы.

Для добавления новой задачи необходимо нажать в области списка задач правую кнопку мыши (ПКМ) и выбрать пункт "*Добавить*". При этом имя задачи инициализируется автоматически уникальным числом, пользователь должен ввести только время выполнения и период. Для удаления необходимо выбрать нужные задачи из списка, нажать ПКМ и выбрать пункт "*Удалить*".

По результатам моделирования выводится временная диаграмма, количество вытеснений и пропущенных крайних сроков для каждой задачи, а также фактический и расчетный коэффициенты загрузки процессора для введенного набора задач. Количество вытеснений – это счетчик, показывающий, сколько раз за время моделирования данная задача вытеснялась задачами с более высоким приоритетом. Количество пропущено крайних сроков – это счетчик, показывающий, сколько раз за время моделирования данная задача пропустила свой крайний срок выполнения.

Использование процессора задачами отображается синей линией, различные события отображаются стрелками.

 Задача перешла в состояние готовности

 Выполнение задачи завершилось

 Задача была вытеснена

 Задача пропустила свой крайний срок выполнения

Подробные сведения о событии можно получить путем установки указателя мыши на соответствующий элемент графика.

В ходе моделирования программа ведет запись всех событий в журнал, которых хранится в файле “log-<date>.txt”. На основе журнала и временной диаграммы можно по шагам проверить работу выбранного алгоритма планирования.

3. Порядок выполнения работы

В таблице 9 приведены варианты заданий. Для указанных исходных данных выполнить указанные действия для алгоритмов RMS и EMS.

1) в ручном режиме:

- рассчитать коэффициент использования процессора для заданной группы задач;
- построить временную диаграмму выполнения заданной группы задач (период планирования от 0 до 50 мс);
- из построенной диаграммы определить время простоя и коэффициент использования процессора;
- сравнить расчетный и реальный коэффициенты использования процессора;
- описать, на каких тактах происходило вытеснение задач;
- при возникновении ситуации «deadlock» указать момент времени, когда эта ситуация произошла;

2) запустить эмулятор планировщика MultiPlanner и построить временную диаграмму выполнения заданной группы задач, сверить полученные в п.1 результаты с программой;

3) изменить исходные данные задач таким образом, чтобы заданная группа задач стала не планируемой для алгоритма RMS, построить новую временную диаграмму в программе MultiPlanner, прокомментировать полученные результаты.

4) сравнить временные диаграммы алгоритмов RMS и EDF для измененных данных.

5) выполнить аналогичные действия с эмулятором MultiPlanner_v2.

Таблица 9

	Вариант 1		Вариант 2		Вариант 3		Вариант 4		Вариант 5	
	C, мс	T, мс	C, мс	T, мс	C, мс	T, мс	C, мс	T, мс	C, мс	T, мс
з.1	2	10	3	7	3	6	2	10	1	12
з.2	3	15	1	10	3	8	3	15	6	16
з.3	5	20	1	8	1	10	9	20	2	20
з.4	1	7	-	-	-	-	-	-	-	-

	Вариант 6		Вариант 7		Вариант 8		Вариант 9		Вариант 10	
	C, мс	T, мс	C, мс	T, мс	C, мс	T, мс	C, мс	T, мс	C, мс	T, мс
з.1	3	20	1	8	7	17	2	7	2	5
з.2	1	8	3	12	2	10	3	8	1	11
з.3	4	15	3	20	1	9	1	9	3	11
з.4	1	11	3	16	-	-	-	-	-	-

Список источников

1. Небезопасный код, типы указателей и указатели функций. [Электронный ресурс] https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/unsafe-code.

2. Кобылянский В.Г. Системы реального времени. Операционная система QNX. Методические указания к выполнению лабораторных работ. – Новосибирск: Изд-во НГТУ, 2017

3. Кобылянский В. Г. Системы реального времени: учебное пособие. –Новосибирск: Изд-во НГТУ, 2015. –88 стр

4. Гриценко Ю. Б. Системы реального времени : учебное пособие / Ю. Б. Гриценко. — Москва : ТУСУР, 2017. — 253 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/110216 (дата обращения: 22.10.2021). — Режим доступа: для авториз. пользователей.

5. Алексеев Д., Ведревич Е., Волков А., Горошко Е., Горчак М., Жавнис Р., Крисак А., Сошин Д., Цилюрик О., Чиликин А. Практика работы с QNX. – М.: Издательский дом “КомБук”, 2004.

6. Зыль С.Н. Операционная система реального времени QNX: от теории к практике. – СПб.: БХВ-Петербург, 2004.

7. Зыль С.Н. QNX Momentics: Основы применения. – СПб.: БХВ-Петербург, 2005.

8. Робачевский А.М., Немнюгин С.А., Стесик О.Л. Операционная система UNIX. – СПб.: БХВ-Петербург, 2005.

9. Встроенная документация QNX 6.

10. Цилюрик О., Горошко Е. QNX/UNIX: анатомия параллелизма. – СПб.: Символ-Плюс, 2006.

11. Операционная система реального времени QNX Neutrino 6.3. Системная архитектура: Пер. с англ. – СПб.: БХВ-Петербург, 2006.

12. Кёртен Р. Введение в QNX Neutrino 2. Руководство для разработчиков приложений реального времени. – СПб.: БХВ-Петербург, 2005.