Лабораторная работа №1-4. Формальные языки, грамматики и их свойства

1. Дана грамматика. Постройте вывод заданной цепочки:

a) $S \to T \mid T{+}S \mid T{-}S$
$T \to F \mid F{*}T$
$F \to a \mid b$
Цепочка $a{-}b{*}a{+}b$

b) $S \to aSBC \mid abC$
$CB \to BC$
$bB \to bb$
$bC \to bc$
$cC \to cc$
Цепочка $aaabbbccc$

2. Какой язык порождается грамматикой с правилами:

a) $S \to aaCFD$
$AD \to D$
$F \to AFB \mid AB$
$Cb \to bC$
$AB \to bBA$
$CB \to C$
$Ab \to bA$
$bCD \to \varepsilon$

б) $S \to A\perp \mid B\perp$
$A \to a \mid Ba$
$B \to b \mid Bb \mid Ab$

3. Построить грамматику, порождающую язык:

a) $\mathbf{L} = \{a^n b^m c^k \mid n, m, k > 0\}$
б) $\mathbf{L} = \{0^n (10)^m \mid n, m \geq 0\}$
в) $\mathbf{L} = \{a_1 a_2 \ldots a_n a_n \ldots a_2 a_1 \mid a_i \in \{0, 1\}\}$

4. К какому типу по Хомскому относится грамматика с правилами:

a) $S \to 0A1 \mid 01$
$0A \to 00A1$
$A \to 01$

б) $S \to Ab$
$A \to Aa \mid ba$

5. Эквивалентны ли грамматики с правилами:

$S \to aSL \mid aL$
$L \to Kc$
$cK \to Kc$
$K \to b$

и

$S \to aSBc \mid abc$
$cB \to Bc$
$bB \to bb$

6. Построить КС-грамматику, эквивалентную грамматике с правилами:

$S \to AB \mid ABS$
$AB \to BA$
$BA \to AB$
$A \to a$
$B \to b$

7. Построить регулярную грамматику, эквивалентную грамматике с правилами:

$S \to A\,.\,A$
$A \to B \mid BA$
$B \to 0 \mid 1$

8. Постройте контекстно-свободную грамматику для:

а) выражения *while* в языке Си;
б) выражения *for* в языке Си;
в) выражения *do-while* в языке Си.

9. Дана грамматика **G**:

$$S \to aSbS \mid bSaS \mid \varepsilon$$

а) Постройте все возможные деревья вывода для цепочки *abab*.
б) Является ли эта грамматика неоднозначной?

10. Напишите регулярное выражение для:

а) множества идентификаторов, где идентификатор – это последовательность букв или цифр, начинающаяся с буквы или '_';
б) множества вещественных констант с плавающей точкой, состоящих из целой части, десятичной точки, дробной части, символа e или E, целого показателя степени с необязательным знаком и необязательного суффикса типа – одной из букв f, F, l или L. Целая и дробная части состоят из последовательностей цифр. Может отсутствовать либо целая, либо дробная часть (но не обе сразу).

11. Написать леволинейную регулярную грамматику, эквивалентную данной праволинейной, допускающую детерминированный разбор:

а) $S \to 0S \mid 0B$
$B \to 1B \mid 1C$
$C \to 1C \mid \perp$

б) $S \to aA \mid aB \mid bA$
$A \to bS$
$B \to aS \mid bB \mid \perp$

12. Даны две грамматики **G1** и **G2**, порождающие языки **L1** и **L2**. Построить регулярную грамматику для $\mathbf{L1} \cap \mathbf{L2}$. Для полученной грамматики построить детерминированный конечный автомат.

G1: $S \to S1 \mid A0$
$A \to A1 \mid 0$

G2: $S \to A1 \mid B0 \mid E1$
$A \to S1$
$B \to C1 \mid D1$
$C \to 0$
$D \to B1$
$E \to E0 \mid 1$

Контрольные вопросы

1. Дайте определение цепочки, языка. Что такое синтаксис и семантика языка?
2. Какие существуют методы задания языков? Какие дополнительные вопросы необходимо решить при задании языка программирования?
3. Что такое грамматика? Дайте определения грамматики.
4. Как выглядит описание грамматики в форме Бэкуса-Наура?
5. Какие типы грамматик существуют?
6. Какие грамматики относятся к регулярным грамматикам?
7. Можно ли для языка, заданного леволинейной грамматикой, построить праволинейную грамматику, задающую эквивалентный язык?
8. Всякая ли регулярная грамматика является однозначной?

9. В чём заключается отличие автоматных грамматик от других регулярных грамматик? Всякая ли регулярная грамматика является автоматной?
10. Что такое конечный автомат (КА)? Дайте определение детерминированного и недетерминированного конечных автоматов.

Список литературы

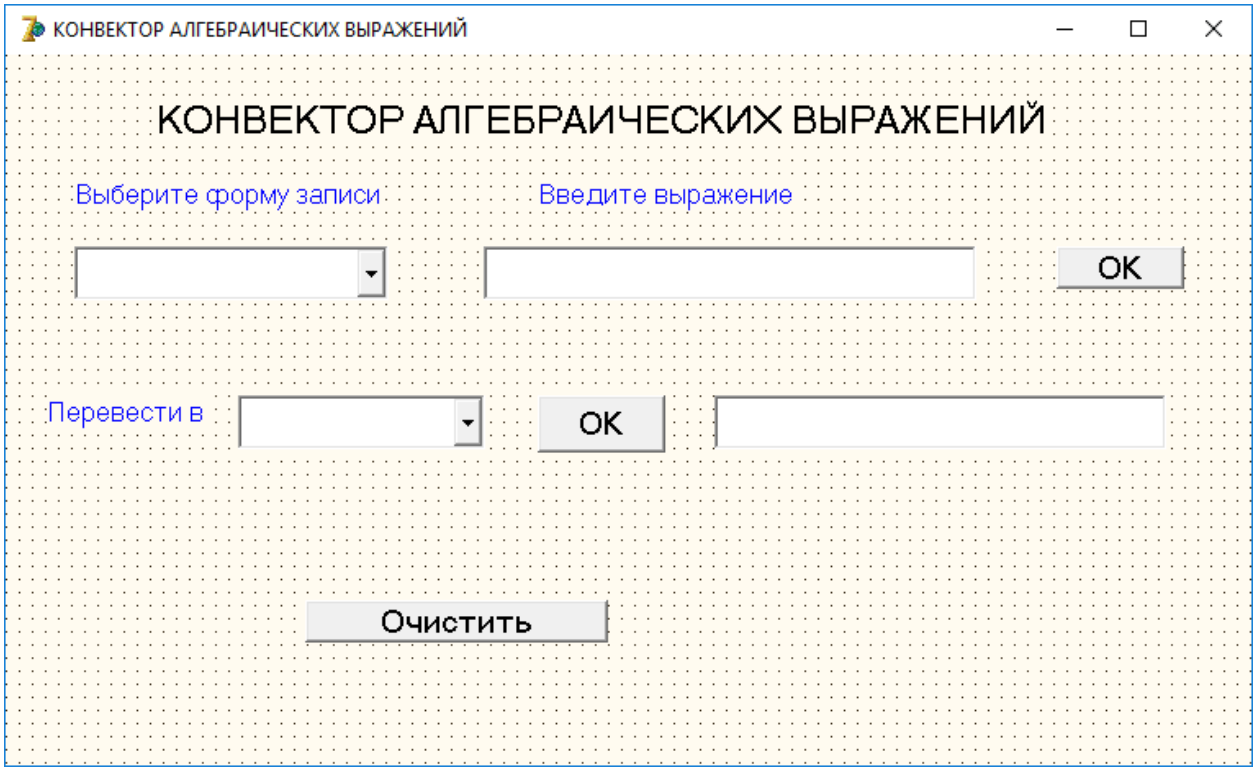
1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1: Синтаксический анализ: Пер. с англ. – М. : Мир, 1978. – 613 с.
2. Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Д. Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. – М. : Издательский дом «Вильямс», 2002. – 528 с.
3. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб.: Питер, 2010. – 400 с.
4. Cooper K.D., Torczon L. Engineering a Compiler, 2nd ed. – Elsevier, Inc., 2012. – 825 p.

Лабораторные работы 5-7
Регулярные языки и конечные автоматы

Задание

Разработать программу преобразования: инфиксной ↔ префиксной ↔ постфиксной формы записи алгебраических выражений.

Примерный вид программы:



Преобразование инфиксного скобочного выражения в постфиксную запись

```
Const OperSet:set of char= ['+','-','*','/','(',')','^'];

Var Stack : array [1..10] of Char;
    InputString : String;
    OutputString : String;
    I,J,K, TopStack : Word;
    Symb : Char;
    SymbTmp : Char;

Function ChToInt(C:Char):Integer;
Var Result : Integer;
Begin
     Case C Of
          '(': Result := 0;
```

```
        ')': Result := 1;
        '-': Result := 2;
        '+': Result := 2;
        '*': Result := 3;
        '/': Result := 3;
        '^': Result := 4;
    End;
    ChToInt := Result;
End;

Function PRCD(Ch1, Ch2 : Char):Boolean;
Var Result : Boolean;
Begin
    If ChToInt(Ch1) >= ChToInt(Ch2) Then Result := True
                                    Else Result := False;

    If Ch2='(' Then Result := False;
    If Ch1='(' Then Result := False;
    If (Ch2=')') And (Ch1<>'(') Then Result := True;
    If (Ch2='(') And (Ch1=')') Then Result := False;

    PRCD := Result;
End;

Function Empty:Boolean;
Var Result : Boolean;
Begin
    If TopStack = 0 Then Result := True
                    Else Result := False;
    Empty := Result;
End;

Function TopStackSymb:Char;
Begin
    TopStackSymb := Stack[TopStack];
End;

Procedure Push(Ch : Char);
Begin
    Inc(TopStack);
    Stack[TopStack] := Ch;
End;

Function Pop:Char;
Begin
```

```
    Pop := Stack[TopStack];
    Stack[TopStack] := ' ';
    Dec(TopStack);
End;

Begin
    ClrScr;
    TopStack := 0;
    OutputString := '';

    ReadLn(InputString);

    For I := 1 To 10 Do Stack[I] := ' ';

    For I := 1 To Length(InputString) Do Begin
       Symb := InputString[I];
       If Symb IN OperSet
         Then Begin
            While Not(Empty) And PRCD(TopStackSymb,Symb) Do Begin
                SymbTmp := Pop;
                OutputString := OutputString + SymbTmp;
            End;
            If (Empty) Or (Symb<>')') Then Push(Symb)
                                Else SymbTmp := Pop;

         End Else OutputString := OutputString + Symb;
    End;

    While Not(Empty) Do Begin
        SymbTmp := Pop;
        OutputString := OutputString + SymbTmp;
    End;

    WriteLn(InputString);
    WriteLn(OutputString);
    ReadLn;
End.
```

Преобразование инфиксного скобочного выражения в префиксную запись

```
Const OperSet:set of char= ['+','-','*','/','(',')','^'];

Var Stack : array [1..10] of Char;
   InputString : String;
```

```
   OutputString : String;
   I,J,K, TopStack : Word;
   Symb : Char;
   SymbTmp : Char;


Function ChToInt(C:Char):Integer;
Var Result : Integer;
Begin
    Case C Of
        '(': Result := 0;
        ')': Result := 1;
        '-': Result := 2;
        '+': Result := 2;
        '*': Result := 3;
        '/': Result := 3;
        '^': Result := 4;
    End;
    ChToInt := Result;
End;

Function PRCD(Ch1, Ch2 : Char):Boolean;
Var Result : Boolean;
Begin
    If ChToInt(Ch1) >= ChToInt(Ch2) Then Result := True
                            Else Result := False;

    If Ch2='(' Then Result := False;
    If Ch1='(' Then Result := False;
    If (Ch2=')') And (Ch1<>'(') Then Result := True;
    If (Ch2='(') And (Ch1=')') Then Result := False;

    PRCD := Result;
End;

Function Empty:Boolean;
Var Result : Boolean;
Begin
    If TopStack = 0 Then Result := True
                Else Result := False;
    Empty := Result;
End;

Function TopStackSymb:Char;
Begin
```

```
    TopStackSymb := Stack[TopStack];
End;

Procedure Push(Ch : Char);
Begin
    Inc(TopStack);
    Stack[TopStack] := Ch;
End;

Function Pop:Char;
Begin
    Pop := Stack[TopStack];
    Stack[TopStack] := ' ';
    Dec(TopStack);
End;

Function Inverse(St : String):String;
Var
  Temp : String;
  I : Integer;
Begin
    Temp := '';
    For I := 1 to Length(St) Do Begin

        Case St[I] of
        '(' : St[I] := ')';
        ')' : St[I] := '(';
        End;

        Temp := St[I] + Temp;
    End;
    Inverse := Temp;
End;

Begin
    ClrScr;
    TopStack := 0;
    OutputString := '';
    InputString := 'A+(B+C)*D';

    WriteLn(InputString);
    InputString := Inverse(InputString);

    For I := 1 To 10 Do Stack[I] := ' ';
```

```
    For I := 1 To Length(InputString) Do Begin
       Symb := InputString[I];
       If Symb IN OperSet
         Then Begin
            While Not(Empty) And PRCD(TopStackSymb,Symb) Do Begin
                SymbTmp := Pop;
                OutputString := OutputString +SymbTmp;
            End;
            If (Empty) Or (Symb<>')') Then Push(Symb)
                                Else SymbTmp := Pop;

         End Else OutputString := OutputString + Symb;
    End;

    While Not(Empty) Do Begin
        SymbTmp := Pop;
        OutputString :=OutputString + SymbTmp;
    End;

    For I := 0 To TopStack Do Begin
       WriteLn(Stack[I]);
    End;

    OutputString := Inverse(OutputString);
    WriteLn(OutputString);

    ReadLn;
End.
```

Преобразование постфиксного выражения в инфиксную (скобочную) запись

```
Const OperSet:set of char= ['+','-','*','/','(',')','^'];

Type TStack = record
          St : String[20];
          Co : Integer;
    End;

Var Stack : array [1..10] of TStack;
   InputString : String;
   OutputString : String;
   I,J,K, TopStack : Word;
   Symb : Char;
   SymbTmp : Char;
```

```
Function ChToInt(C:Char):Integer;
Var Result : Integer;
Begin
    Case C Of
        '-': Result := 3;
        '+': Result := 3;
        '*': Result := 2;
        '/': Result := 2;
        '^': Result := 1;
    Else Result := 0;
    End;
    ChToInt := Result;
End;

Procedure Push(Ch : Char);
Begin
    Inc(TopStack);
    Stack[TopStack].St := Ch;
    Stack[TopStack].Co := ChToInt(Ch);
End;

Function Max(A,B,C:Integer):Integer;
Var Tmp : Integer;
Begin
    If A > B Then Tmp := A
           Else Tmp := B;
    If Tmp > C Then Max := Tmp
             Else Max := C;
End;

Procedure Pop;
Var Tmp : Integer;
Begin
    Tmp := Max(Stack[TopStack-0].Co,
               Stack[TopStack-1].Co,
               Stack[TopStack-2].Co);

    If Stack[TopStack].Co <> Tmp Then Begin
      If Length(Stack[TopStack-2].St) <> 1 Then
        Stack[TopStack-2].St := '(' + Stack[TopStack-2].St + ')';
      If Length(Stack[TopStack-1].St) <> 1 Then
        Stack[TopStack-1].St := '(' + Stack[TopStack-1].St + ')';
    End;
```

```
    Stack[TopStack-2].Co := Tmp;

    Stack[TopStack-2].St := Stack[TopStack-2].St +
                        Stack[TopStack-0].St +
                        Stack[TopStack-1].St;

    Dec(TopStack,2);
End;

Begin
    ClrScr;
    TopStack := 0;
    OutputString := '';
    ReadLn(InputString);

    OutputString := InputString;

    For I := 1 To Length(InputString) Do Begin
       Push(InputString[I]);
       If Stack[TopStack].Co <> 0 Then Pop;
    End;

    OutputString := Stack[1].St;

    WriteLn(InputString);
    WriteLn(OutputString);
    ReadLn;
End.
```

Преобразование префиксного выражения в инфиксную (скобочную) запись

```
Const OperSet:set of char= ['+','-','*','/','(',')','^'];

Type TStack = record
          St : String[20];
          Co : Integer;
    End;

Var Stack : array [1..10] of TStack;
   InputString : String;
   OutputString : String;
   I,J,K, TopStack : Word;
   Symb : Char;
   SymbTmp : Char;
```

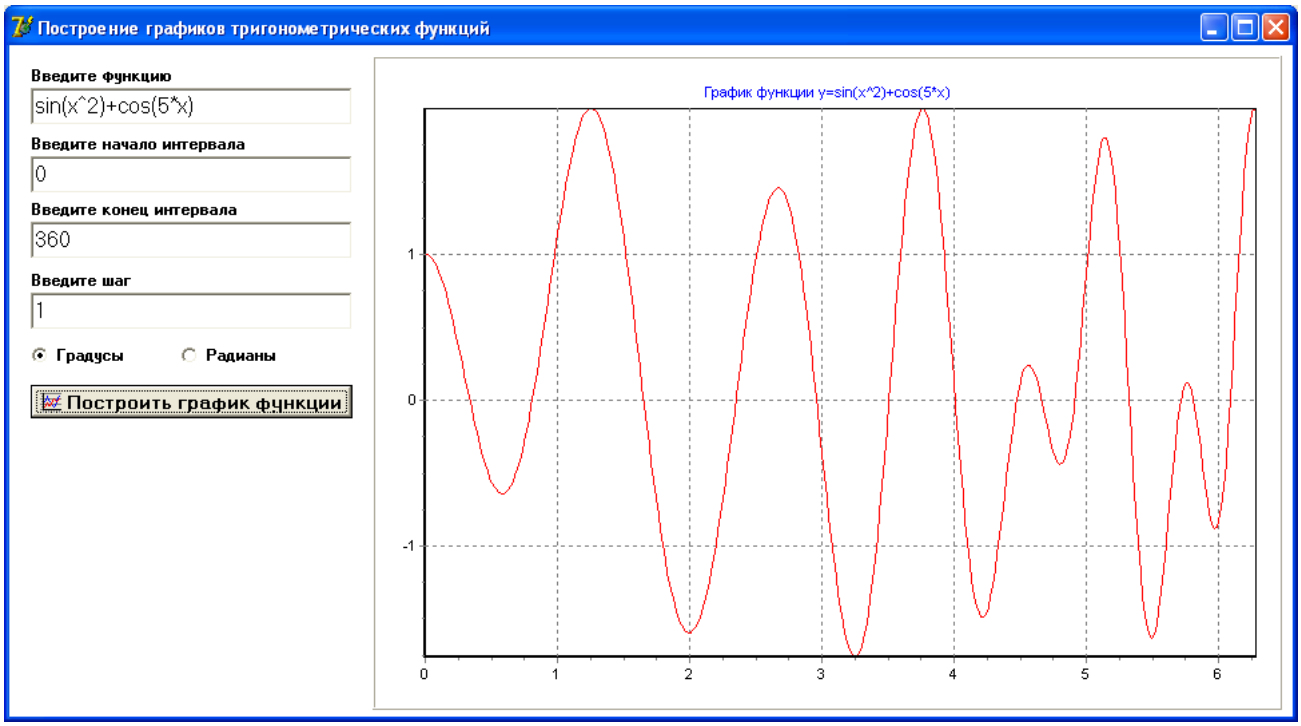
```
Function ChToInt(C:Char):Integer;
Var Result : Integer;
Begin
    Case C Of
        '-': Result := 3;
        '+': Result := 3;
        '*': Result := 2;
        '/': Result := 2;
        '^': Result := 1;
    Else Result := 0;
    End;
    ChToInt := Result;
End;

Procedure Push(Ch : Char);
Begin
    Inc(TopStack);
    Stack[TopStack].St := Ch;
    Stack[TopStack].Co := ChToInt(Ch);
End;

Function Max(A,B,C:Integer):Integer;
Var Tmp : Integer;
Begin
    If A > B Then Tmp := A
           Else Tmp := B;
    If Tmp > C Then Max := Tmp
             Else Max := C;
End;

Procedure Pop;
Var Tmp : Integer;
Begin
    Tmp := Max(Stack[TopStack-0].Co,
               Stack[TopStack-1].Co,
               Stack[TopStack-2].Co);

    If Stack[TopStack].Co <> Tmp Then Begin
      If Length(Stack[TopStack-2].St) <> 1 Then
        Stack[TopStack-2].St := '(' + Stack[TopStack-2].St + ')';
      If Length(Stack[TopStack-1].St) <> 1 Then
        Stack[TopStack-1].St := '(' + Stack[TopStack-1].St + ')';
    End;
```

```
    Stack[TopStack-2].Co := Tmp;

    Stack[TopStack-2].St := Stack[TopStack-2].St +
                        Stack[TopStack-0].St +
                        Stack[TopStack-1].St;

    Dec(TopStack,2);
End;

Function Inverse(St : String):String;
Var
  Temp : String;
  I : Integer;
Begin
    Temp := '';
    For I := 1 to Length(St) Do Begin

       Case St[I] of
       '(' : St[I] := ')';
       ')' : St[I] := '(';
       End;

       Temp := St[I] + Temp;
    End;
    Inverse := Temp;
End;

Begin
    ClrScr;
    TopStack := 0;
    OutputString := '';
    ReadLn(InputString);
    WriteLn(InputString);
    InputString := Inverse(InputString);

    For I := 1 To Length(InputString) Do Begin
       Push(InputString[I]);
       If Stack[TopStack].Co <> 0 Then Pop;
    End;

    OutputString := Stack[1].St;
    OutputString := Inverse(OutputString);
    WriteLn(OutputString);
    ReadLn;
End.
```

Лабораторные работы 8-11

Контекстно-свободные языки и автоматы с магазинной памятью.

Программа «ПОСТРОЕНИЕ ГРАФИКОВ ФУНКЦИЙ» позволяет строить и сохранять графики функций одной переменной. При открытии программы появляется окно, в котором уже по умолчанию установлена функция f(x)= sin(x^2)+cos(5*x) на отрезке от 0 до 360 градусов с шагом 1 градус. При необходимости можно изменить функцию и параметры. После этого нажимаем кнопку «Построить график функции».



Построенный график функции

Список поддерживаемых функций и операций:

cos(x) - косинус

sin(x) - синус

tg(x) = tan(x) - тангенс

ctg(x) - котангенс

abs(x) = fabs(x) - модуль

exp(x) - экспонента

ln(x) - натуральный логарифм

x+y - сложение x с y

x-y - вычитание из x y

x*y - умножение x на y

x/y - деление x на y

x^y - возведение x в степень y

Для того чтобы лучше понять работу программы приведем некоторые коды:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var a,b,h,x,y:real; f:boolean;
begin
Series1.Clear;  // очистка графика
Chart1.Title.Text.Clear; //очистка заголовка графика
Chart1.Legend.Visible:=false; //убираем легенду
Chart1.Title.Text.Add('График функции y='+LabeledEdit1.Text); //добавляем в заголовок графика введенную функцию
a:=strtofloat(LabeledEdit2.Text); // получаем начало диапазона
b:=strtofloat(LabeledEdit3.Text); //получаем конец диапазона
h:=strtofloat(LabeledEdit4.Text); //получаем шаг построения графика

if RadioButton1.Checked then begin a:=a*Pi/180; b:=b*Pi/180; h:=h*Pi/180 end; //проверяем в градусах или радианах введены данные и преобразуем их при необходимости
x:=a; //начальное значение аргумента функции
While x<=b do begin  //начало цикла построения графика функции
y:=LatsCalculate(x,0,LabeledEdit1.Text,f); //вызов функции из модуля для интерпретации функции и вычисления ее значения
Series1.AddXY(x,y,'',clRed); // добавление точки в грфик
x:=x+h; //наращивание аргумента на шаг
end;
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Chart1.Title.Text.Clear; //очистка заголовка функции при создании формы
end;
```

Модуль для интерпретации и вычисления функции:

```
unit MyLib;

interface

uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Menus, Math;

  function IsPoint (s:string):string;
  function IsDigit(Ch:Char):Boolean;
  function IsX(ch:char):boolean;
  function IsSign(Ch:Char):Boolean;
  function IsSeparator(Ch:Char):Boolean;
  function Factor(const S:string;var P:Integer):Extended;
  function IsExponent(Ch:Char):Boolean;
  function Number(const S:string;var P:Integer):Extended;
  function Base(const S:string;var P:Integer):Extended;
  function IsOperator(Ch:Char):Boolean;
  function IsOperator1(Ch:Char):Boolean;
  function IsOperator2(Ch:Char):Boolean;
  function Calculate(const S:string):Extended;
  function LatsCalculate (const x1,y1:real; s:string; var b:boolean):Extended;
```

```
type ESyntaxError=class(Exception);

var x,y:real;
    division:boolean;

implementation

// Проверка символа на соответствие
function IsDigit(Ch:Char):Boolean;
 begin
  Result:=(Ch in ['0'..'9']) or (ch='x');
 end;

function IsX(ch:char):boolean;
begin
 result:= (upcase(ch)='X');
end;

Function IsY (ch:char):boolean;
begin
 result:= upcase (ch)='Y';
end;
// Проверка символа на соответствие
function IsSign(Ch:Char):Boolean;
 begin
  Result:=(Ch='+') or (Ch='-')
 end;

// Проверка символа на соответствие
```

```
function IsSeparator(Ch:Char):Boolean;
 begin
  Result:=Ch=',';
 end;

function IsPoint (s:string):string;
var i:integer;
begin
 for i:=1 to length(s) do
  if s[i]='.' then s[i]:=',';
  Result:=s;
end;



// Проверка символа на соответствие
function IsExponent(Ch:Char):Boolean;
 begin
  Result:=(Ch='E') or (Ch='e')
 end;


   function Number(const S:string;var P:Integer):Extended;
 var InitPos:Integer;
  begin
   // InitPos нам понадобиться для выделения подстроки,
   // которая будет передана в StrToFloat
   InitPos:=P;
   if (P<=Length(S)) and IsSign(S[P]) then
   Inc(P);
```

```
if (not IsX(s[p]) )and not IsY(s[p]) then begin

// После символа может быть переменная
if (P>Length(S)) or not IsDigit(S[P]) then
 raise ESyntaxError.Create('Ожидается цифра в позиции '+IntToStr(P));
repeat
 Inc(P)
until (P>Length(S)) or not IsDigit(S[P]);
if (P<=Length(S)) and IsSeparator(S[P]) then
 begin
  Inc(P);
  if (P>Length(S)) or not IsDigit(S[P]) then
   raise ESyntaxError.Create('Ожидается цифра в позиции '+IntToStr(P));
  repeat
   Inc(P)
  until (P>Length(S)) or not IsDigit(S[P]);
 end;
if (P<=Length(S)) and IsExponent(S[P]) then
 begin
  Inc(P);
  if P>Length(S) then
   raise ESyntaxError.Create('Неожиданный конец строки');
  if IsSign(S[P]) then
   Inc(P);
  if (P>Length(S)) or not IsDigit(S[P]) then
   raise ESyntaxError.Create('Ожидается цифра в позиции '+IntToStr(P));
  repeat
   Inc(P)
  until (P>Length(S)) or not IsDigit(S[P]);
 end;
```

```
 Result:=StrToFloat(Copy(S,InitPos,P-InitPos)) end
 else if IsX (s[p]) then begin Result:=x; inc(p); end
 else if IsY (s[p]) then begin result:=y; inc(p);  end
 else raise ESyntaxError.Create('Некорректный символ в позиции '+IntToStr(P));
 end;

// Проверка символа на соответствие
function IsOperator(Ch:Char):Boolean;
 begin
  Result:=Ch in ['+','-','*','/']
 end;

// Проверка строки на соответствие
// и вычисление выражения
 // Проверка символа на соответствие
function IsOperator1(Ch:Char):Boolean;
 begin
  Result:=Ch in ['+','-']
 end;

// Проверка символа на соответствие
function IsOperator2(Ch:Char):Boolean;
 begin
  Result:=Ch in ['*','/']
 end;

 function Expr(const S:string;var P:Integer):Extended;
 forward;
```

```
function Term(const S:string;var P:Integer):real;
 var OpSymb:Char;
 res:real;
  begin
   division:=false;
   Result:=Factor(S,P);
   while (P<=Length(S)) and IsOperator2(S[P]) do
    begin
     OpSymb:=S[P];
     Inc(P);
     res:=Factor(S,P);
     if opsymb='*' then Result:=Result*res;
     if (opsymb='/') and (res<>0) then Result:=Result/res
     else if (opsymb='/') and (res=0) then begin  result:=1000; division:=true;
end;
    end
  end;

// Выделение подстроки, соответствующей ,
// и её вычисление
function Expr(const S:string;var P:Integer):Extended;
 var OpSymb:Char;
  begin
   Result:=Term(S,P);
   while (P<=Length(S)) and IsOperator1(S[P]) do
    begin
     OpSymb:=S[P];
     Inc(P);
     case OpSymb of
```

```
      '+':Result:=Result+Term(S,P);
      '-':Result:=Result-Term(S,P)
     end
    end
  end;

// Вычисление выражения

// Вычисление функции, имя которой передаётся через FuncName
function Func(const FuncName,S:string;var P:Integer):Extended;
 var Arg:Extended;
  begin
   // Вычисляем аргумент
   Arg:=Expr(S,P);
   division:=false;
   // Сравниваем имя функции с одним из допустимых
   if AnsiCompareText(FuncName,'sin')=0 then
     Result:=Sin(Arg)
   else if AnsiCompareText(FuncName,'cos')=0 then
    Result:=Cos(Arg)
    else if AnsiCompareText(FuncName,'tg')=0 then
    Result:=Sin(Arg)/Cos(Arg)
    else if AnsiCompareText(FuncName,'ctg')=0 then
    Result:=Cos(Arg)/Sin(Arg)
   else if (AnsiCompareText(FuncName,'ln')=0) and (arg>0) then
    Result:=Ln(Arg)
   else if (AnsiCompareText (FuncName, 'ln')=0) and (arg<=0) then
   begin
   division:=true;
   result:=0; end
```

```
  else
   raise ESyntaxError.Create('Неизвестная функция '+FuncName)
  end;

// Выделение из строки идентификатора и определение,
// является ли он переменной или функцией
function Identifier(const S:string;var P:Integer):Extended;
 var InitP:Integer;
     IDStr,VarValue:string;
  begin
   // Запоминаем начало идентификатора
   InitP:=P;
   // Первый символ был проверен ещё в функции Base.
   // Сразу переходим к следующему
   Inc(P);
   while (P<=Length(S)) and (S[P] in ['A'..'Z','a'..'z','_','0'..'9']) do
    Inc(P);
   // Выделяем идентификатор из строки
   IDStr:=Copy(S,InitP,P-InitP);
   // Если за ним стоит открывающая скобка - это функция
   if (P<=Length(S)) and (S[P]='(') then
    begin
     Inc(P);
     Result:=Func(IDStr,S,P);
     // Проверяем, что скобка закрыта
     if (P>Length(S)) or (S[P]<>')') then
      raise ESyntaxError.Create('Ожидается ")" в позиции '+IntToStr(P));
     Inc(P)
    end
   // если скобки нет - переменная
```

```
            else
             begin
               if AnsiCompareText(IDStr,'x')=0 then result:=x else
               if AnsiCompareText(IDStr,'y')=0 then result:=y else
               if AnsiCompareText(IDStr,'pi')=0 then result:=pi else
                raise ESyntaxError.Create('Некорректный символ в позиции: '+IntToStr(P));
             end
            end;

          // Выделение подстроки, соответствующей ,
          // и её вычисление
          function Base(const S:string;var P:Integer):Extended;
           begin
            if P>Length(S) then
             raise ESyntaxError.Create('Неожиданный конец строки');
            // По первому символу подстроки определяем,
            // какое это основание
            case S[P] of
             '(': // выражение в скобках
              begin
               Inc(P);
               Result:=Expr(S,P);
               // Проверяем, что скобка закрыта
               if (P>Length(S)) or (S[P]<>')') then
                raise ESyntaxError.Create('Ожидается ")" в позиции '+IntToStr(P));
               Inc(P)
              end;
             '0'..'9': // Числовая константа
              Result:=Number(S,P);
```

```
          'A'..'Z','a'..'z','_': // Идентификатор (переменная или функция)
           Result:=Identifier(S,P)
          else
           raise ESyntaxError.Create('Некорректный символ в позиции
'+IntToStr(P))
          end
         end;

        // Выделение подстроки, соответствующей ,
        // и её вычисление
        function Factor(const S:string;var P:Integer):Extended;
         begin
          if P>Length(S) then
           raise ESyntaxError.Create('Неожиданный конец строки');
          // По первому символу подстроки определяем,
          // какой это множитель
          case S[P] of
           '+': // унарный "+"
            begin
             Inc(P);
             Result:=Factor(S,P)
            end;
           '-': // унарный "-"
            begin
             Inc(P);
             Result:=-Factor(S,P)
            end
          else
           begin
            Result:=Base(S,P);
```

```
        if (P<=Length(S)) and (S[P]='^') then
         begin
          Inc(P);
          Result:=Power(Result,Factor(S,P))
         end
        end
       end
      end;

    function Calculate(const S:string):Extended;
     var P:Integer;
      begin
       P:=1;
       Result:=Expr(S,P);
       if P<=Length(S) then
        raise ESyntaxError.Create('Некорректный символ в позиции
'+IntToStr(P))
       end;

    function LatsCalculate (const x1,y1:real; s:string; var b:boolean):Extended;
    begin
    b:=false;
    x:=x1;
    y:=y1;
    Result:=Calculate(S);
    b:=division;
    end;

    end.
```

Задание

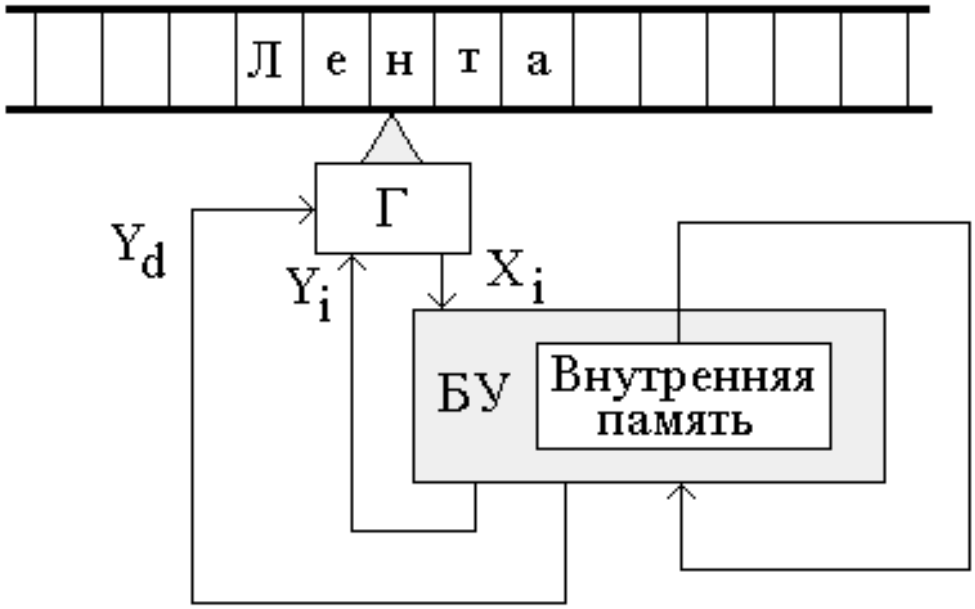
Добавить функции:

a. log2(x) – логарифм по основанию 2;
b. log(x) – логарифм по основанию 10;
c. arcsin(x) – арксинус;
d. arccos(x) – арккосинус;
e. arctg(x) - арктангенс;
f. arcctg(x) – арккотангенс.

Лабораторные работы 12-15

Свободные языки и машина Тьюринга

Машина Тьюринга представляет собой устройство для выполнения алгоритмов преобразования информации. В теории алгоритмов машина Тьюринга используется как средство для описания алгоритмов. Считается, что если алгоритм решения некоторой задачи существует, то этот алгоритм может быть реализован на машине Тьюринга, т.е. для этого алгоритма может быть построена машина Тьюринга. С точки зрения теории цифровых автоматов машина Тьюринга (более строго - ее блок управления) представляет собой универсальный преобразователь информации.



Машина Тьюринга (МТ) может иметь структуру, показанную на рис.1.1.

Рис.1.1.

В состав машины входят:

- ☐ лента, на которой записаны исходные данные и на которую записываются результаты решения задачи;
- ☐ головка записи/чтения информации (Г);
- ☐ блок управления (БУ).

Лента состоит из отдельных ячеек. В каждую ячейку может быть записан символ из некоторого алфавита. На ленту предварительно записывается исходная информация. В процессе работы МТ с ленты с помощью головки считывается символ, находящийся над головкой (текущий символ X_i). При считывании информация в ячейке стирается. После считывания символа X_i в результате работы машины на данном шаге на ленту вместо Xi записывается новый символ Y_i. Лента считается

бесконечной в одну или обе стороны. Лента является внешней памятью машины.

Головка служит для чтения и записи информации. Головка с помощью привода может перемещаться вдоль ленты вправо или влево на одну ячейку на каждом шаге. В каждый момент времени для записи или чтения доступна только одна ячейка ленты.

Блок управления организует работу машины в целом. Он анализирует считываемую информацию и управляет записью символов Y_i на ленту, а также перемещением головки. Блок управления имеет внутреннюю память. Информация во внутренней памяти представляет собой состояние машины Тьюринга. Реакция машины на считанный символ X_i зависит не только от значения этого символа, но и от состояния машины. Состояние машины на каждом шаге работы машины может изменяться. Новое состояние машины определяется значением символа X_i и старым состоянием машины.

Перед началом работы на ленту наносится исходная информация. Головка устанавливается под ячейкой ленты , в которой записан первый символ. Машина переводится в начальное состояние.

Процесс преобразования информации в машине Тьюринга состоит из отдельных шагов. На каждом шаге машина выполняет следующие элементарные операции:

- ☐ чтение символа X_i из ячейки, под которой размещена головка;
- ☐ анализ считанного символа в соответствии с алгоритмом решения задачи;
- ☐ запись в ячейку вместо символа X_i нового символа Y_i (он может совпадать с X_i);
- ☐ перемещение головки на одну ячейку влево или вправо;
- ☐ переход машины в новое состояние (запись новой информации во внутреннюю память).

На каждом шаге работы значение символа Y_i, направление перемещения головки и новое состояние машины зависят от значения символа X_i и текущего состояния машины. Поэтому процесс работы машины Тьюринга может быть описан в виде совокупности команд, каждая из которых имеет следующий вид:

$$X_iQ_i \rightarrow Y_iY_dQ_j,$$

где X_i - символ, считанный с ленты;

Q_i - текущее состояние машины;

Y_i - символ, записываемый на ленту;

Y_d = (П, Л, Ст) - сигнал управления движением головки (П - сигнал движения головки вправо; Л - сигнал движения головки влево; Ст - сигнал останова);

Q_j - новое состояние машины.

Алгоритм работы машины Тьюринга может быть описан с помощью ориентированного графа. При этом вершины графа соответствуют состояниям машины, а дуги указывают переходы из одного состояния в другое. На дугах графа отмечаются входные символы X_i и соответствующие им сигналы Y_i и Y_d.

В качестве примера на рис.1.2 показан граф машины Тьюринга, которая обрабатывает информацию на ленте, составленную из букв А и В. Массив обрабатываемой информации ограничен слева и справа символами-разделителями *.

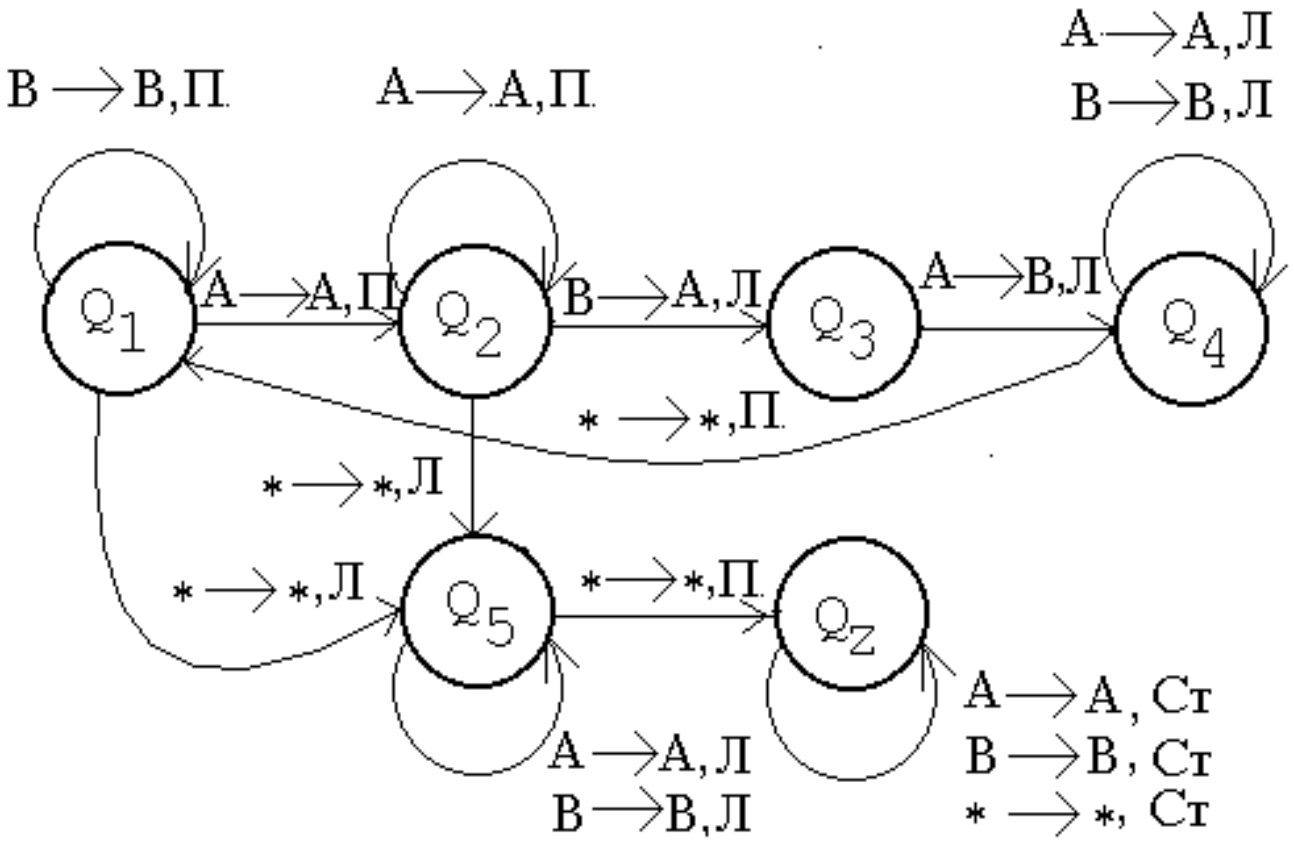


Рис.1.2.

Перед началом работы головка устанавливается справа от левого символа*, а машина переводится в начальное состояние Q_1.

Суть обработки заключается в том, что машина просматривает информацию на ленте, последовательно перемещая головку вправо, и при обнаружении пары символов АВ заменяет ее на пару ВА (выполняет подстановку АВ →ВА). Решение задачи заканчивается, если на ленте не останется ни одной комбинации символов вида АВ.

Особенность задачи заключается в том, что в результате такой подстановки на ленте могут появляться новые комбинации, которых ранее на ленте не было. Поэтому принят алгоритм, при котором после каждой подстановки головка возвращается в исходное положение и просмотр ленты

повторяется. В этом случае признаком отсутствия на ленте комбинаций типа АВ является достижение головки при её движении вправо ячейки, в которой записан символ *. Тогда головка перемещается влево в исходное положение, и машина останавливается, переходя в конечное состояние Q_Z. Для примера на рис.1.3 показан вид ленты с исходной и преобразованной информацией.

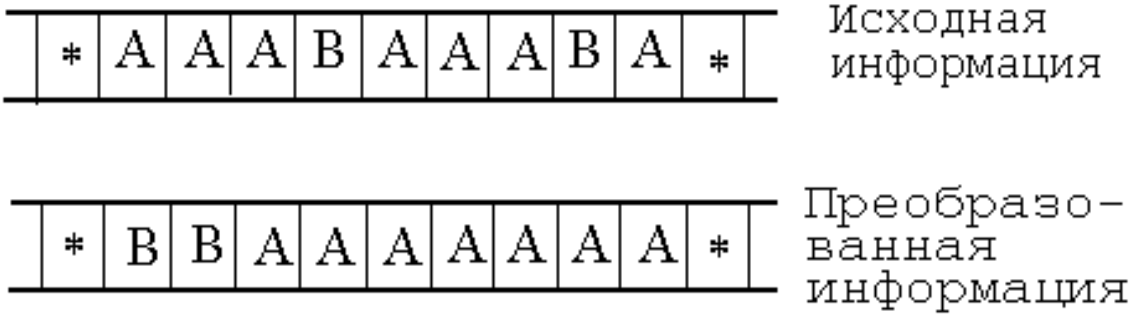


Рис.1.3

Каждое состояние на графе рис.1.2 имеет свои особенности. Переход машины из одного состояния в другое происходит в случае считывания определенной информации и этот факт необходимо запомнить. В данном случае состояния можно определить следующим образом:

- состояние Q_1 - исходное состояние. В этом состоянии машина перемещает головку вправо до обнаружения символа А;
- состояние Q_2 - состояние, в которое машина переходит, если при движении головки вправо последний считанный символ был символом А. В состоянии Q_2 машина "ждет" символ В;
- состояние Q_3 - состояние, в которое машина переходит при обнаружении комбинации вида АВ. В этом состоянии символ В заменяется на символ А и головка смещается влево на один шаг;
- состояние Q_4 - состояние, в которое машина переходит после замены всей комбинации АВ на комбинацию ВА. В этом состоянии машины головка перемещается влево в исходное положение;
- состояние Q_5 - состояние, в которое машина переходит, если при движении головки вправо из исходного положения на ленте не обнаружено ни одной комбинации вида АВ. В этом состоянии машины головка

перемещается

влево в исходное положение;

☐ состояние Q_z - конечное состояние, в которое машина переходит, если закончена обработка информации на ленте и головка установлена в исход-

ное положение.

Алгоритм работы машины Тьюринга может быть описан также в табличной форме. Для этой цели используется таблица переходов и выходов машины.

В этой таблице строки соответствуют состояниям Q_i, а столбцы - входным сигналам X_i. На пересечении строки Q_i и столбца X_i записываются выходные символы Y_i, Y_d и Q_i.

Таблица переходов и выходов машины, граф работы которой показан на рис. 1.2, имеет вид табл.1.1.

Таблица 1

Состояния МТ Q_i	Входные символы X_i		
	A	B	*
Q_1	A, П, Q2	B, П, Q1	*, Л, Q5
Q_2	A, П, Q2	A, Л, Q3	*, Л, Q5
Q_3	B, Л, Q4	——	——
Q_4	A, Л, Q4	B, Л, Q4	*, П, Q1
Q_5	A, Л, Q5	B, Л, Q5	*, П, Qz
Q_z	A, Ст, Qz	B, Ст, Qz	*, Ст, Qz

Рассматриваемая машина Тьюринга работает следующим образом. Из исходного положения головка перемещается вправо до обнаружения пары символов AB. При появлении такой комбинации головка перемещается влево и производится последовательная замена символа B на символ A, затем символа A на символ B. После замены головка перемещается влево до символа *, затем делает шаг вправо и устанавливается в исходное положение.

Далее начинается новый цикл поиска комбинации AB при движении головки вправо. Работа МТ продолжается до тех пор, пока на ленте не останется ни одной пары символов AB. Признаком этого является то, что головка при движении вправо доходит до символа *. Тогда машина переходит в состояние Q_5, в котором организуется возвращение головки в исходное положение. Затем машина переходит в состояние Q_z, головка останавливается, и работа машины заканчивается. Результат работы машины

остается в виде информации на ленте (рис. 1.3)

Подготовка к выполнению работы

При подготовке к выполнению работы необходимо:

Задание 1.

- ✓ Изучить теоретическую часть.
- ✓ Составить граф и таблицу переходов и выходов машины Тьюринга, которая просматривает информацию на ленте и выполняет подстановки (задания в таблице 2).
- ✓ Составить схему алгоритма имитационного моделирования работы машины Тьюринга.

Таблица 2

№ ЗАД	**Подстановка**	**№ ЗАД**	**Подстановка**	**№ ЗАД**	**Подстановка**	**№ ЗАД**	**Подстановка**
1	100-101	2	AAA-BBB	3	ABB-ABA	4	111-000
5	AAA-BBB	6	AAB-BBB	7	111-011	8	110-000
9	AAB-BAA	10	ABA-BBB	11	110-011	12	101-000
13	ABA-BAA	14	ABB-BBB	15	101-011	16	100-000
17	ABB-BAA	18	BAA-ABB	19	100-011	20	011-100

- ✓ Составить программу моделирования, имитирующую процесс работы машины Тьюринга.
- ✓ Подготовить материалы для отчета.

Задание 2.

Решить задачи, используя программу моделирования работы машины Тьюринга.

Пример: Реализовать функцию f(n)=n+1 в десятичной системе счисления.

	Q_1	Q_2
0	0 → Q_1	1 ↓ ⛔
1	1 → Q_1	2 ↓ ⛔
2	2 → Q_1	3 ↓ ⛔
3	3 → Q_1	4 ↓ ⛔
4	4 → Q_1	5 ↓ ⛔
5	5 → Q_1	6 ↓ ⛔
6	6 → Q_1	7 ↓ ⛔
7	7 → Q_1	8 ↓ ⛔
8	8 → Q_1	9 ↓ ⛔
9	9 → Q_1	0 ← Q_2
⊔	_ ← Q_2	1 ↓ ⛔

	Q1	Q2
0	0>1	1.0
1	1>1	2.0
2	2>1	3.0
3	3>1	4.0
4	4>1	5.0
5	5>1	6.0
6	6>1	7.0
7	7>1	8.0
8	8>1	9.0
9	9>1	0<2
_	_<2	1.0

1. Уменьшить число, записанное в двоичной системе счисления, на 1. Каретка стоит над числом.

2. Реализовать функцию f(n)=n+4 в десятичной системе счисления, где n любое десятичное число.
3. Реализовать функцию f(n)=n+25 в десятичной системе счисления, где n любое десятичное число.
4. Удалить все вхождения символа 'a'. Каретка находится на первом символе слова.

Рекомендации по моделированию

При моделировании лента обычно представляется в виде одномерного массива символьных элементов. Количество элементов должно быть не менее 10. Обрабатываемая информация должна быть ограничена слева и справа символами- ограничителями. В качестве таких символов можно использовать любые символы, отличающиеся от записанных во входном слове.

Перемещение головки влево и вправо имитируется изменением индекса элемента массива (влево - уменьшение на единицу, вправо - увеличение).

Логику работы машины Тьюринга удобно программировать по таблице

При выводе результатов необходимо предусмотреть печать исходного массива, промежуточных результатов после каждой замены и окончательного результата работы машины.

Программа на языке Паскаль, моделирующая работу машины Тьюринга в соответствии с табл. 1.1, показана ниже.

```
program MT;

VAR J,I:INTEGER;
    S:STRING;
    F:TEXT;
LABEL Q0,Q1,Q2,Q3,Q4,QZ;
BEGIN
    assign(f,'D:\dat.txt');
    rewrite(f);
    WRITELN('ВВЕДИТЕ ИСХОДНУЮ ПОСЛЕДОВАТЕЛЬНОСТЬ');
    WRITELN(F,'ВВЕДИТЕ ИСХОДНУЮ ПОСЛЕДОВАТЕЛЬНОСТЬ');
    READLN(S);

   WRITELN(F,S);

    WRITELN(F);
    WRITELN (F,'ПОСЛЕДОВАТЕЛЬНОСТЬ ВВЕДЕНА');
```

```
    WRITELN (F,'ПРОЦЕСС РАБОТЫ МТ');
    I:=2;
    Q0: IF S[I] = 'A' THEN BEGIN INC(I); GOTO Q1; END;
        IF S[I] = 'B' THEN BEGIN INC(I); GOTO Q0;
END;
        IF S[I] = '*' THEN BEGIN DEC(I); GOTO Q4;
END;
    Q1: IF S[I] = 'A' THEN BEGIN INC(I); GOTO Q1; END;
        IF S[I] = 'B' THEN BEGIN S[I]:='A'; DEC(I);
GOTO Q2;
                                                 END;
          IF S[I] = '*' THEN BEGIN DEC(I); GOTO Q4;
END;
    Q2:                            S[I]:='B'; DEC(I);
       J:=I;
       WRITELN(F,S);
                          WRITELN(F);
       I:=J;
    Q3: IF S[I] = 'A' THEN BEGIN DEC(I); GOTO Q3; END;
        IF S[I] = 'B' THEN BEGIN DEC(I); GOTO Q3;
END;
        IF S[I] = '*' THEN BEGIN INC(I); GOTO Q0;
END;
    Q4: IF S[I] = 'A' THEN BEGIN DEC(I); GOTO Q4; END;
        IF S[I] = 'B' THEN BEGIN DEC(I); GOTO Q4;
END;
        IF S[I] = '*' THEN BEGIN INC(I); GOTO QZ;
END;
    QZ: WRITELN (F,'РЕЗУЛЬТАТ РАБОТЫ МТ');
       WRITELN(F,S);
     CLOSE(F);
END.
```

Полученные при работе программы результаты имеют следующий вид:

ВВЕДИТЕ ИСХОДНУЮ ПОСЛЕДОВАТЕЛЬНОСТЬ
AABAABBAA
ПОСЛЕДОВАТЕЛЬНОСТЬ ВВЕДЕНА
ПРОЦЕСС РАБОТЫ МТ
ABAAABBAA
BAAAABBAA
BAAABABAA
BAABAABAA
BABAAABAA

BBAAAABAA
BBAAABAAA
BBAABAAAA
BBABAAAAA
BBBAAAAAA
РЕЗУЛЬТАТ РАБОТЫ МТ
BBBAAAAAA

Содержание отчета

В отчет по лабораторной работе включить следующие материалы:

- ☞ Содержание задания.
- ☞ Граф работы машины Тьюринга.
- ☞ Таблица переходов и выходов машины Тьюринга.
- ☞ Схема алгоритма моделирования.
- ☞ Программа моделирования на алгоритмическом языке.
- ☞ Результаты работы машины Тьюринга.
- ☞ Анализ полученных результатов.

ЛАБОРАТОРНО-ПРАКТИЧЕСКАЯ РАБОТА 16

ПЛАНИРОВАНИЕ ИСПОЛЬЗОВАНИЯ ПРОЦЕССОРА. НЕВЫТЕСНЯЮЩАЯ МНОГОЗАДАЧНОСТЬ

Цель работы: формирование умений и навыков подбора невытесняющих алгоритмов планирования на основе анализа их основных характеристик.

Техническое обеспечение: ПЭВМ на базе Intel-80486.

Программное обеспечение: OS Windows XP.

Время выполнения: 2 часа.

1. Краткие теоретические сведения

Планирование использования процессорного времени выступает в качестве краткосрочного планирования. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых стремятся достичь, используя планирование.

Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Далее будем использовать следующие обозначения:

- CPU burst – промежуток времени непрерывного использования процессора;
- I/O burst – промежуток времени непрерывного ожидания ввода-вывода.

Планировщик может принимать решения о выборе для исполнения нового процесса, из числа находящихся в состоянии ***готовности***, в следующих четырех случаях:

1) когда процесс переводится из состояния ***исполнение*** в состояние ***завершение***;

2) когда процесс переводится из состояния ***исполнение*** в состояние ***ожидание***;

3) когда процесс переводится из состояния ***исполнение*** в состояние ***готовность*** (например, после прерывания от таймера);

4) когда процесс переводится из состояния ***ожидание*** в состояние ***готовность*** (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии ***исполнение***, не может дальше исполняться, и для выполнения всегда необходимо выбрать новый процесс. Если планирование осуществляется только в случаях 1 и 2, то говорят, что имеет место *невытесняющее (nonpreemptive) планирование*. В противном случае говорят о *вытесняющем (preemptive) планировании*.

1.1. First Come, First Served (FCFS)

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First Come, First Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии ***готовность***, организованы в очередь. Когда процесс переходит в состояние ***готовность***, он, а точнее ссылка на его PCB (process control bloct), помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование FIFO – сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения своего текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

1.2. Shortest Job First (SJF)

В том случае, когда известно время следующих CPU burst для процессов, находящихся в состоянии ***готовность***, можно выбрать для исполнения процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название «кратчайшая работа первой» или Shortest Job First.

SJF алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим.

При невытесняющем SJF планировании процессор предоставляется избранному процессу на все требующееся ему время, независимо от событий, происходящих в вычислительной системе.

1.3. Приоритетное планирование

При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим.

Если приоритеты процессов не изменялись с течением времени, то такие приоритеты принято называть статическими.

Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов.

2. Ход работы

2.1. Изучите теоретические сведения.

2.2. Рассмотрите примеры:

Пример 1

Тема: алгоритм планирования FCFS.

Пусть в состоянии ***готовность*** находятся три процесса p_0, p_1 и p_2, для которых известны времена их очередных CPU burst. Эти времена приведены в табл. 1.1 в условных единицах.

Таблица 1.1

Процесс	p_0	p_1	p_2
Продолжительность очередного CPU burst	13	4	1

Для простоты будем полагать: вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, процессы не совершают операций ввода-вывода, и время переключения контекста пренебрежимо мало.

Пусть процессы расположены в очереди процессов готовых к исполнению в порядке p_0, p_1, p_2. Тогда картина их выполнения выглядит как показано на рис. 1.1.

Рис. 1.1. Выполнение процессов при порядке p_0, p_1, p_2

Первым для выполнения выбирается процесс p_0, который получает процессор на все время своего CPU burst, то есть на 13 единиц времени. После его окончания в состояние ***исполнение*** переводится процесс p_1, занимая процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p_2.

Время ожидания для процесса p_0 составляет 0 единиц времени, для процесса p_1 – 13 единиц, для процесса p_2 – 13 + 4 = = 17 единиц.

Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени.

Полное время выполнения для процесса p_0 составляет 13 единиц времени, для процесса p_1 – 13 + 4 = 17 единиц, для процесса p_2 – 13 + 4 + 1 = 18 единиц.

Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.

Если те же самые процессы расположены в порядке p_2, p_1, p_0, то картина их выполнения будет соответствовать рис. 1.2.

Рис. 1.2. Выполнение процессов при порядке p_2, p_1, p_0

Время ожидания для процесса p_0 равняется 5 единицам времени, для процесса p_1 – 1 единице, для процесса p_2 – 0 единиц.

Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае.

Полное время выполнения для процесса p_0 получается равным 18 единицам времени, для процесса p_1 – 5 единицам, для процесса p_2 – 1 единице.

Среднее полное время выполнения составляет (18 + 5 + 1)/3 = = 8 единиц времени, что почти в 2,7 раза меньше чем при первой расстановке процессов.

Как видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди.

Пример 2

Тема: алгоритм приоритетного планирования (невытесняющий режим).

Пусть в очередь процессов, находящихся в состоянии ***готовность***, поступают процессы p_0, p_1, p_2 и p_3 с различными временами CPU burst и различными моментами их появления в очереди (табл. 1.2).

Таблица 1.2

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p_0	0	6	4
p_1	2	2	3
p_2	6	7	2
p_3	0	5	1

В вычислительных системах не существует определенного соглашения, какое значение приоритета – 1 или 4 – считать более приоритетным. Во избежание путаницы, во всех последующих примерах будем предполагать, что большее значение соответствует более низкому приоритету, то есть наиболее приоритетным в нашем примере является процесс p_3, а наименее приоритетным – процесс p_0.

Первым для выполнения в момент времени t = 0 выбирается процесс p_3, как обладающий наивысшим приоритетом.

После его завершения в момент времени t = 5 в очереди процессов, готовых к исполнению, окажутся два процесса p_0 и p_1. Наивысший приоритет из них у процесса p_1, он и начнет выполняться (табл. 1.3).

Таблица 1.3

Время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г
p_1			Г	Г	Г	И	И			
p_2							Г	И	И	И
p_3	И	И	И	И	И					
Время	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	И	И	И	И	И	И
p_1										
p_2	И	И	И	И						
p_3										

Используем следующие обозначения:

И – для процесса, находящегося в состоянии ***исполнение***,

Г – для процесса в состоянии ***готовность***,

пустые ячейки соответствуют завершившимся процессам.

Затем в момент времени t = 8 для исполнения будет избран процесс p_2 и лишь потом – процесс p_0.

2.3. Выполните задания.

1. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	2	4
2	1	7
3	6	5
4	4	1
5	0	4

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для алгоритма планирования FCFS. При вычислениях считать, что процессы не совершают операций ввода-вывода, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления.

2. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	6	4
2	2	8
3	10	6
4	0	5
5	4	2

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для невытесняющего алгоритма планирования SJF. При вычислениях считать, что процессы не совершают операций ввода-вывода, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления.

3. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	4	3
2	2	4
3	3	6
4	1	4
5	0	5

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для каждого из двух алгоритмов планирования FCFS и SJF. При вычислениях считать, что процессы не совершают операций ввода-вывода, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления.

4. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	2	3
2	5	1
3	0	6
4	3	2
5	1	2

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для каждого из двух алгоритмов планирования FCFS и SJF. При вычислениях считать, что процессы не совершают операций ввода-вывода, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления.

Контрольные вопросы

1. Какие алгоритмы планирования вы знаете?
2. Каковы критерии планирования?
3. Какие требования предъявляются к алгоритмам планирования?
4. Что такое многоуровневые очереди?
5. Какова цель приоритетного планирования?

Содержание отчета

1. Цель работы.
2. Краткое описание алгоритмов планирования.
3. Табличное выполнение заданий.
4. Выводы по работе.

ЛАБОРАТОРНО-ПРАКТИЧЕСКАЯ РАБОТА 17

ПЛАНИРОВАНИЕ ИСПОЛЬЗОВАНИЯ ПРОЦЕССОРА. ВЫТЕСНЯЮЩАЯ МНОГОЗАДАЧНОСТЬ

Цель работы: формирование умений и навыков подбора вытесняющих алгоритмов планирования на основе анализа их основных характеристик.

Техническое обеспечение: ПЭВМ на базе Intel-80486.

Программное обеспечение: OS Windows XP.

Время выполнения: 2 часа.

1. Краткие теоретические сведения

Когда процесс переводится из состояния ***исполнение*** в состояние ***готовность*** (например, после прерывания от таймера) и когда процесс переводится из состояния ***ожидание*** в состояние ***готовность*** (завершилась операция ввода-вывода или произошло другое событие), то говорят, что имеет место *вытесняющее* (*preemptive*) *планирование*. Термин «вытесняющее планирование» возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния ***исполнение*** другим процессом.

1.1. Round Robin (RR)

Самый простой и часто используемый алгоритм планирования является модификацией алгоритма FCFS и реализован в режиме вытесняющего планирования.

Каждому процессу предоставляется квант времени процессора. Когда квант заканчивается, процесс переводится планировщиком в конец очереди. При блокировке процесс выпадает из очереди.

Этот алгоритм получил название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени. Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени.

При выполнении процесса возможны два варианта:

1) время непрерывного использования процессора, требующееся процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение выбирается новый процесс из начала очереди и таймер начинает отсчет кванта заново;

2) продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

1.2. Shortest Job First (SJF) вытесняющий

SJF алгоритм краткосрочного планирования может быть так же и вытесняющим.

При вытесняющем SJF планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания времени очередного CPU burst для исполняющихся процессов. При краткосрочном планировании можно делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса.

2. Ход работы

2.1. Изучите теоретические сведения.

2.2. Рассмотрите примеры:

Пример 1

Тема: алгоритм планирования RR.

Рассмотрим пример 1 (см. описание практической работы 1, с. 6–8) с порядком процессов p_0, p_1, p_2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется табл. 2.1.

Таблица 2.1

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	И	И	И	Г	Г	Г	Г	Г	И	И	И	И	И	И	И	И	
p_1	Г	Г	Г	Г	И	И	И	И										
p_2	Г	Г	Г	Г	Г	Г	Г	Г	И									

Используем следующие обозначения:

И – для процесса, находящегося в состоянии ***исполнение***,

Г – для процесса в состоянии ***готовность***,

пустые ячейки соответствуют завершившимся процессам.

Состояния процессов показаны на определенном интервале времени, то есть колонка с номером 1 соответствует промежутку времени от 0 до 1.

Первым для исполнения выбирается процесс p_0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, то есть в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p_1, p_2, p_0.

Следующим начинает выполняться процесс p_1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения.

Теперь очередь процессов в состоянии ***готовность*** состоит из двух процессов p_2, p_0. Процессор выделяется процессу p_2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p_0 – единственному, не закончившему к этому моменту свою работу.

Время ожидания для процесса p_0 (количество символов Г в соответствующей строке) составляет 5 единиц времени, для процесса p_1 – 4 единицы времени, для процесса p_2 – 8 единиц времени.

Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6 единицам времени.

Полное время выполнения для процесса p_0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p_1 – 8 единиц, для процесса p_2 – 9 единиц.

Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6 единицам времени.

Легко видеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p_0, p_1, p_2 для величины кванта времени равной 1 (табл. 2.2).

Таблица 2.2

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	Г	Г	И	Г	И	Г	И	Г	И	И	И	И	И	И	И	И	И
p_1	Г	И	Г	Г	И	Г	И	Г	И									
p_2	Г	Г	И															

Время ожидания для процесса p_0 составит 5 единиц времени, для процесса p_1 – тоже 5 единиц, для процесса p_2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени.

Среднее полное время исполнения составит (18 + 9 + 3)/3 = = 10 единиц времени.

Пример 2

Тема: вытесняющий алгоритм планирования SJF.

Для рассмотрения примера вытесняющего SJF планирования мы возьмем ряд процессов p_0, p_1, p_2 и p_3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (табл. 2.3).

Таблица 2.3

Процесс	Время появления в очереди	Продолжительность очередного CPU burst
p_0	0	6
p_1	2	2
p_2	6	7
p_3	0	5

В начальный момент времени в состоянии ***готовность*** находятся только два процесса p_0 и p_3. Меньшее время очередного CPU burst оказывается у процесса p_3, поэтому он и выбирается для исполнения (табл. 2.4).

По прошествии 2-х единиц времени в систему поступает процесс p_1. Время его CPU burst меньше, чем остаток CPU burst у процесса p_3, который вытесняется из состояния ***исполнение*** и переводится в состояние ***готовность***.

Таблица 2.4

Время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	И	И	И
p_1			И	И						
p_2							Г	Г	Г	Г
p_3	И	И	Г	Г	И	И	И			
Время	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
p_0	И	И	И							
p_1										
p_2										
p_3	Г	Г	Г	И	И	И	И	И	И	И

По прошествии еще 2-х единиц времени процесс p_1 завершается, и для исполнения вновь выбирается процесс p_3.

В момент времени t = 6 в очереди процессов, готовых к исполнению, появляется процесс p_2, но поскольку ему для работы нужно 7 единиц времени, а процессу p_3 осталось трудиться всего 2 единицы времени, то процесс p_3 остается в состоянии ***исполнение***. После его завершения в момент времени t = 7 в очереди находятся процессы p_0 и p_2, из которых выбирается процесс p_0.

Последним получит возможность выполняться процесс p_2.

2.3. Выполните задания.

1. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	2	2
2	5	2
3	4	3
4	7	1
5	0	4

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для алгоритма планирования RR. При вычислениях считать, что процессы не совершают операций ввода-вывода, величину кванта времени принять равной 1, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления. Вновь

прибывший процесс помещается в начало очереди процессов, готовых к исполнению, и, следовательно, сразу выбирается на исполнение.

2. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	6	4
2	2	8
3	10	6
4	0	5
5	4	2

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для вытесняющего алгоритма планирования SJF. При вычислениях считать, что процессы не совершают операций ввода-вывода, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления.

3. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	4	3
2	2	4
3	3	6
4	1	4
5	0	5

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для каждого из трех алгоритмов планирования FCFS, RR и SJF. При вычислениях считать, что процессы не совершают операций ввода-вывода, величину кванта времени принять равной 1, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления. Для алгоритма RR принять, что вновь прибывший процесс помещается в начало очереди процессов, готовых к исполнению, и, следовательно, сразу выбирается на исполнение.

4. Пусть в вычислительную систему поступают пять процессов различной длительности по следующей схеме:

Номер процесса	Время поступления	Время исполнения
1	2	3
2	5	1
3	0	6
4	3	2
5	1	2

Вычислите среднее время между стартом процесса и его завершением (*turnaroud time*) и среднее время ожидания процесса (*waiting time*) для каждого из трех алгоритмов планирования FCFS, RR и SJF. При вычислениях считать, что процессы не совершают операций ввода-вывода, величину кванта времени принять равной 1, временем переключения контекста пренебречь. Процесс, поступающий в систему, считать готовым к исполнению в момент поступления. Для алгоритма RR принять, что вновь прибывший процесс помещается в начало очереди процессов, готовых к исполнению, и, следовательно, сразу выбирается на исполнение.

Контрольные вопросы

1. В чем суть вытесняющего планирования?
2. На что может влиять размер выделяемого процессу кванта времени?
3. Как можно сгруппировать алгоритмы планирования?

Содержание отчета

1. Цель работы.
2. Краткое описание алгоритмов планирования.
3. Табличное выполнение заданий.
4. Выводы по работе.

ЛАБОРАТОРНО-ПРАКТИЧЕСКАЯ РАБОТА 18
МОДЕЛИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ПРОЦЕССОВ С ПОМОЩЬЮ СЕТЕЙ ПЕТРИ

Цель работы: приобретение навыков моделирования последовательных процессов с помощью сетей Петри.

Техническое обеспечение: ПЭВМ на базе Intel-80486.

Программное обеспечение: OS Windows XP.

Время выполнения: 2 часа.

1. Краткие теоретические сведения

Сети Петри – это инструмент для математического моделирования и исследования сложных систем. Цель представления системы в виде сети Петри и последующего анализа этой сети состоит в получении важной информации о структуре и динамическом поведении моделируемой системы. Эта информация может использоваться для оценки моделируемой системы и выработки предложений по ее усовершенствованию.

Сети Петри предназначены для моделирования систем, которые состоят из множества взаимодействующих друг с другом *компонент*. При этом компонента сама может быть системой. Действиям различных компонент системы присущ параллелизм.

Примерами таких систем могут служить вычислительные системы, в том числе и параллельные, компьютерные сети, программные системы, обеспечивающие их функционирование, а также экономические системы, системы управления дорожным движением и т. д.

В одном из подходов к проектированию и анализу систем сети Петри используются как вспомогательный инструмент анализа. Здесь для построения системы применяются общепринятые методы проектирования. Затем построенная система моделируется сетью Петри, и модель анализируется. Если в ходе анализа в проекте найдены изъяны, то с целью их устранения проект модифицируется. Модифицированный проект затем снова моделируется и анализируется. Этот цикл повторяется до тех пор, пока проводимый анализ не приведет к успеху.

Другой подход предполагает построение проекта сразу в виде сети Петри. Методы анализа применяются только для создания проекта, не содержащего ошибок. Затем сеть Петри преобразуется в реальную рабочую систему.

В первом случае необходима разработка методов моделирования систем сетями Петри, а во втором случае должны быть разработаны методы реализации сетей Петри системами.

1.1. Теоретико-множественное определение сетей Петри

Пусть *мультимножество* – это множество, допускающее вхождение нескольких экземпляров одного и того же элемента.

Сеть Петри N является четверкой $N = (P, T, I, O)$, где

$P = \{p_1, p_2, ..., p_n\}$ – конечное множество *позиций*, $n \geq 0$;

$T = \{t_1, t_2, ..., t_m\}$ – конечное множество *переходов*, $m \geq 0$;

$I: T \rightarrow P^*$ – входная функция, сопоставляющая переходу мультимножество его входных позиций;

$O: T \rightarrow P^*$ – выходная функция, сопоставляющая переходу мультимножество его выходных позиций.

Позиция $p \in P$ называется *входом* для перехода $t \in T$, если $p \in I(t)$. Позиция $p \in P$ называется *выходом* для перехода $t \in T$, если $p \in O(t)$.

Структура сети Петри определяется ее позициями, переходами, входной и выходной функциями.

Использование мультимножеств входных и выходных позиций перехода, а не множеств, позволяет позиции быть *кратным входом* и *кратным выходом* перехода соответственно. При этом *кратность* определяется числом экземпляров позиции в соответствующем мультимножестве.

1.2. Графы сетей Петри

Наиболее наглядным представлением сети Петри является графическое – двудольный, ориентированный мультиграф.

Граф сети Петри обладает двумя типами узлов:

1) *кружок* μ, представляющий позицию сети Петри;

2) *планка* — (или ▭), представляющая переход сети Петри.

Ориентированные дуги этого графа (стрелки) соединяют переход с его входными и выходными позициями. При этом дуги направлены от входных позиций к переходу и от перехода к выходным позициям. Кратным входным и выходным позициям перехода соответствуют кратные входные и выходные дуги.

П р и м е ч а н и е. В графе сети Петри невозможны дуги между двумя позициями и между двумя переходами.

1.3. Маркировка сетей Петри

Маркировка – это размещение по позициям сети Петри *фишек*, изображаемых на графе сети Петри точками. Фишки используются для определения выполнения сети Петри. Количество фишек в позиции при выполнении сети Петри может изменяться от 0 до бесконечности.

Маркировка μ сети Петри $N = (P, T, I, O)$ есть функция, отображающая множество позиций P во множество Nat неотрицательных целых чисел. Маркировка μ может быть так же опреде-

лена как n-вектор $\mu = <\mu(p_1), \mu(p_2), \ldots, \mu(p_n)>$, где n – число позиций в сети Петри для каждого $1 \le I \le n\mu(p_i) \in Nat$ количество фишек в позиции p_i.

Маркированная сеть Петри $N = (P, T, I, O, \mu)$ определяется совокупностью структуры сети Петри (P, T, I, O) и маркировки μ. На рис. 3.1 представлена маркированная сеть Петри $\mu = <1, 0, 1>$.

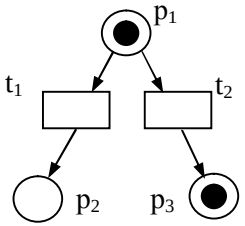


Рис. 3.1. Маркированная сеть Петри $\mu = <1, 0, 1>$.

Множество всех маркировок сети Петри бесконечно. Если фишек, помещаемых в позицию, слишком много, то в кружке этой позиции надо указывать просто их количество.

1.4. Правила выполнения сетей Петри

Сеть Петри *выполняется* посредством *запусков* переходов. *Запуск* перехода управляется фишками в его входных позициях и сопровождается удалением фишек из этих позиций и добавлением новых фишек в его выходные позиции.

Переход может запускаться только в том случае, когда он разрешен. Переход называется *разрешенным*, если каждая из его входных позиций содержит число фишек, не меньшее, чем число дуг, ведущих из этой позиции в переход (или кратности входной дуги).

Сеть Петри до запуска перехода t_1 изображена на рис. 3.2, а, сеть Петри после запуска перехода t_1 – на рис. 3.2, б.

Преобразование маркировки сети Петри изображено на рис. 3.2. Переход t_1 преобразует маркировку $\mu = <5, 1>$ в маркировку $\mu' = <2, 3>$.

Переход t в маркированной сети Петри с маркировкой μ может быть запущен всякий раз, когда он разрешен, и в результате этого запуска образуется новая маркировка μ'.

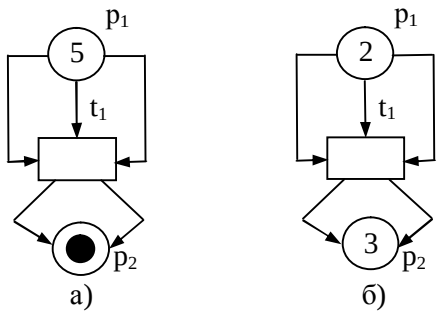


Рис. 3.2. Сеть Петри:
а – до запуска перехода, б – после запуска перехода.

Запуски могут выполняться до тех пор, пока существует хотя бы один разрешенный переход. Когда не останется ни одного разрешенного перехода, выполнение *прекратится.*

Если запуск произвольного перехода t преобразует маркировку μ сети Петри в новую маркировку μ', то будем говорить, что μ' достижима из μ посредством запуска перехода t, и обозначать этот факт, как $\mu \rightarrow t\,\mu'$. Это понятие очевидным образом обобщается для случая последовательности запусков разрешенных переходов. Через $R(N, \mu)$ обозначим множество всех достижимых маркировок из начальной маркировки μ в сети Петри N.

1.5. Моделирование систем на основе сетей Петри

Рассмотрим метод моделирования на основе сетей Петри, а также его применение для моделирования параллельных систем взаимодействующих процессов и решения ряда классических задач из области синхронизации процессов.

Представление системы сетью Петри основано на двух основополагающих понятиях: *событиях* и *условиях*. Возникновением событий управляет состояние системы, которое может быть описано множеством условий. Условие может принимать значение либо «истина», либо «ложь».

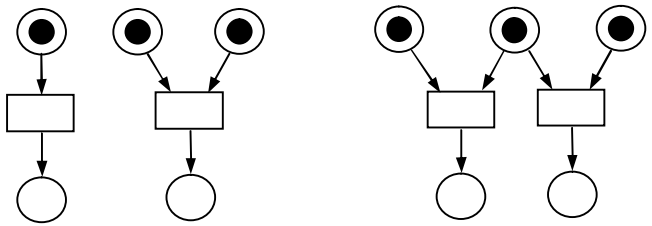
Возникновение события в системе возможно, если выполняются определенные условия – *предусловия* события. Возникновение события может привести к выполнению других условий – *постусловий* события. В качестве примера рассмотрим следующую ниже задачу моделирования.

1.6. Моделирование одновременного (параллельного) возникновения независимых событий системы в сети Петри

Особенность сетей Петри – их *асинхронная* природа. В сетях Петри отсутствует измерение времени. В них учитывается лишь важнейшее свойство времени – частичное упорядочение событий.

Выполнение сети Петри (или поведение моделируемой системы) рассматривается здесь как *последовательность* дискретных событий, которая является одной из возможных. Если в какой-то момент времени разрешено более одного перехода, то любой из них может стать «следующим» запускаемым. Переходы в сети Петри, моделирующей некоторую систему, представляют ее *примитивные* события (длительность которых считается равной 0), и в один момент времени может быть запущен только один разрешенный переход.

Моделирование одновременного (параллельного) возникновения независимых событий системы в сети Петри демонстрируется на рис. 3.3, а. В этой ситуации два перехода являются разрешенными и не влияют друг на друга в том смысле, что могут быть запущены один вслед за другим в любом порядке.



а) б)

Рис. 3.3. Моделирование одновременного возникновения сети Петри:
а – независимые события; б – конфликтная ситуация

Другая ситуация приведена на рис. 3.3, б. Эти два разрешенные перехода находятся в *конфликте*, то есть запуск одного из них удаляет фишку из общей входной позиции и тем самым запрещает запуск другого.

Таким образом моделируются взаимоисключающие события системы.

П р и м е ч а н и е. Сети Петри предназначены для моделирования упорядочения действий и потока информации, а не для действительного вычисления самих значений.

2. Ход работы

2.1. Изучите теоретические сведения.
2.2. Рассмотрите примеры:
Пример 1
Тема: граф сети Петри.
Сеть Петри N =(P,T,I,O),
$P=\{p_1, p_2, p_3\}$,
$T=\{t_1, t_2\}$,
$I(t_1)=\{ p_1, p_1, p_2\}$, $O(t_1)=\{p_3\}$,
$I(t_2)=\{ p_1, p_2, p_2\}$, $O(t_2)=\{p_3\}$
На рис. 3.4 представлен граф сети Петри для данного примера.

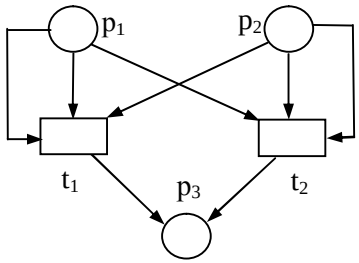


Рис. 3.4. Сеть Петри, иллюстрирующая конкретный пример

Пример 2
Тема: моделирование последовательной обработки запросов сервером базы данных.

Сервер находится в состоянии ожидания до тех пор, пока от пользователя не поступит запрос клиента, который он обрабатывает и отправляет результат такой обработки пользователю.

Условия рассматриваемой системы:
а) сервер ждет;
б) запрос поступил и ждет;
в) сервер обрабатывает запрос;
г) запрос обработан.

События рассматриваемой системы:

1) запрос поступил;

2) сервер начинает обработку запроса;

3) сервер заканчивает обработку запроса;

4) результат обработки отправляется клиенту.

Для перечисленных событий составляется специальная таблица их пред- и постусловий (табл. 3.1).

Таблица 3.1

Событие	Предусловия	Постусловия
1	нет	б
2	а, б	в
3	в	г, а
4	г	нет

В сети Петри условия моделируются позициями, события – переходами. При этом:

- входы перехода являются предусловиями соответствующего события;

- выходы перехода являются постусловиями соответствующего события.

Возникновение события моделируется запуском соответствующего перехода. Выполнение условия представляется фишкой в позиции, соответствующей этому условию. Запуск перехода удаляет фишки, представляющие выполнение предусловий, и образует новые фишки, которые представляют выполнение постусловий.

На рис. 3.5 предусловие выполняется для события 2.

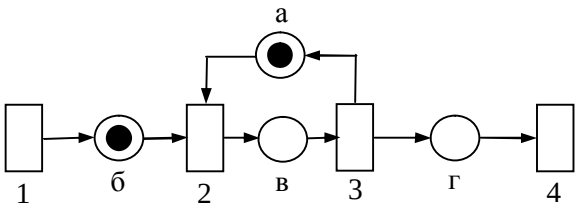


Рис. 3.5. Сеть Петри

2.3. Выполните задания.

1. Разработайте сеть Петри, моделирующую блок-схему, приведенную на рис. 3.6, при этом:

- узлы блок-схемы представить переходами сети Петри;

- дуги блок-схемы – позициями сети Петри;

- фишка в сети Петри должна представить счетчик команд блок-схемы.

Дано: отдельный процесс описан программой на абстрактном языке программирования.

Программа (представляющая два различных аспекта процесса: вычисление и управление) вычисляет Y! и произведение всех четных чисел из отрезка [1, Y] для Y ∈ Nat.

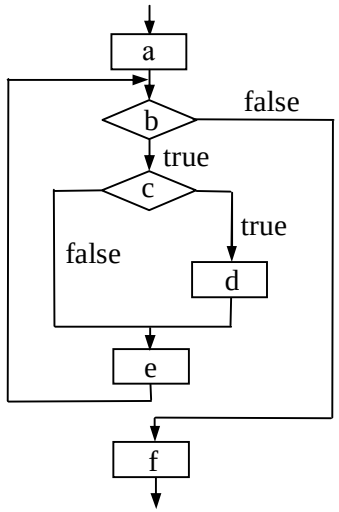


Рис. 3.6. Блок-схема программы

```
begin
    read (Y);
    X1:=1;
    X2:=1;
    while Y>0 do
        begin
        if mod (Y,2)=0
        then begin
                X1:=X1*Y;
                end;
        X2:=X2*Y;
        Y:=Y–1;
        end;
    write(X1);
    write(X2);
end
```

Блоки:
a: read(Y); X1:=!; X2:=1;
b: Y>0
c: mod (Y,2)=0
d: X1:=X1*Y;
e: X2:=X2*Y; Y:=Y–1;
f: write(X1); write(X2);

Блок-схема программы состоит из узлов двух типов:
- принятия решения, обозначаемых ромбами;
- вычисления, обозначаемых прямоугольниками,

а также – дуг между ними.

Контрольные вопросы

1. Какие вы знаете формальные модели для изучения тупиковых ситуаций?
2. Что такое сети Петри?
3. Каковы правила выполнения сетей Петри?
4. Что такое маркировка сетей Петри?
5. В чем заключается срабатывание перехода сети Петри?
6. Что вы знаете об области применения сетей Петри?

Содержание отчета

1. Цель работы.
2. Краткое описание позиций и переходов разрабатываемой модели.
3. Модель последовательного процесса в виде сети Петри.
4. Выводы по работе.

ЛАБОРАТОРНО-ПРАКТИЧЕСКАЯ РАБОТА 19

МОДЕЛИРОВАНИЕ ВЗАИМОДЕЙСТВИЯ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ С ИСПОЛЬЗОВАНИЕМ СЕТЕЙ

Цель работы: приобретение навыков моделирования взаимодействия параллельных процессов с помощью сетей Петри.

Техническое обеспечение: ПЭВМ на базе Intel-80486.

Программное обеспечение: OS Windows XP.

Время выполнения: 4 часа.

1. Краткие теоретические сведения

1.1. Задача взаимного исключения

Задача взаимного исключения является одной из ключевых проблем параллельного программирования. Было предложено много способов решения этой проблемы.

Взаимоисключение одновременного доступа процессов к данным необходимо применить при работе нескольких параллельных процессов с общими данными. При этом участки программ процессов для работы с разделяемыми данными образуют так называемые критические области (секции).

Операционная система может в любой момент вытеснить поток P и подключить к процессору другой, но ни один из потоков, которым нужен занятый потоком P ресурс, не получит процессорное время до тех пор, пока поток P не выйдет за границы критической секции.

1.2. Задача «производитель–потребитель»

Имеется большое число вариантов постановки и решения такой задачи в рамках конкретных операционных систем. В общем случае взаимодействуют несколько процессов с жестко распределенными между ними функциями. Одни процессы вырабатывают сообщения, предназначенные для восприятия и обработки другими процессами.

Процесс, вырабатывающий сообщения, называют *производителем*, а воспринимающий сообщения – *потребителем*.

Процессы взаимодействуют через некоторую обобщенную область памяти, которая по смыслу является критическим ресурсом. В эту область процесс–производитель должен поместить очередное сообщение, а процесс–потребитель должен считывать очередное сообщение.

Общий вид постановки задачи «производитель–потребитель». Необходимо согласовать работу двух процессов при одностороннем (в простейшем случае) обмене сообщениями по мере развития процессов таким образом, чтобы удовлетворить следующим требованиям:

- выполнять требования задачи взаимного исключения по отношению к критическому ресурсу – обобщенной памяти для хранения сообщения;

- учитывать состояние обобщенной области памяти, характеризующей возможность или невозможность посылки (принятия) очередного сообщения;

- попытка процесса–производителя поместить очередное сообщение в область, из которой не было считано предыдущее сообщение процессом–потребителем, должна быть блокирована. Процесс–производитель должен быть:

либо оповещен о невозможности помещения сообщения;

либо переведен в состояние ожидания возможности поместить очередное сообщение через некоторое время в область памяти, по мере ее освобождения;

- попытка процесса–потребителя считать сообщение из области в ситуации, когда процесс–производитель не поместил туда очередного сообщения, должна быть заблокирована. Процесс–потребитель должен быть:

либо оповещен о невозможности считывания сообщения;

либо переведен в состояние ожидания поступления очередного сообщения;

- если используется вариант с ожиданием изменения состояния обобщенной области для хранения сообщений, необходимо обеспечить перевод ожидающих процессов в состояние готовности всякий раз, когда изменится состояние области:

либо процесс–производитель поместит очередное сообщение в область, и оно теперь может быть считано ожидающим процессом–потребителем;

либо процесс–потребитель считал очередное сообщение из области и обеспечил возможность ожидающему процессу–потребителю послать очередное сообщение.

П р и м е ч а н и е. Множественность постановки задачи «производитель–потребитель» определяется следующим:

1) процессов–потребителей и процессов–производителей может быть больше одного;

2) каждый из таких процессов может устанавливать не только одностороннюю, но и двустороннюю связь через одну и ту же обобщенную область или другие области;

3) области могут хранить не только одно, а большое количество сообщений.

1.3. Задача «читатели–писатели»

Существует несколько вариантов этой задачи, однако их основная структура остается неизменной.

Имеется система параллельных процессов, которые взаимодействуют друг с другом следующим образом:

- все процессы делятся на два типа: процессы–читатели и процессы–писатели. Процессы работают с общей областью памяти;

- процессы–читатели считывают, а процессы–писатели записывают информацию в общую область памяти;

- одновременно может быть несколько активных процессов–читателей;

- при записи информации область памяти рассматривается как критический ресурс для всех процессов, то есть, если работает процесс–писатель, то он должен быть единственным активным процессом.

Задача состоит в определении структуры управления, которая не приведет к тупику и не допустит нарушения критерия взаимного исключения.

Наиболее характерная область использования этой задачи – построение файловых систем.

В отношении некоторой области памяти, являющейся по смыслу критическим ресурсом для параллельных процессов, работающих с ней, выделяется два типа процессов:

1) процессы–читатели, которые считывают одновременно информацию из области, если это допускается при работе с конкретным устройством памяти;

2) процессы–писатели, которые записывают информацию в область и могут делать это, только исключая как друг друга, так и процессы–читатели, то есть запись должна удовлетворяться на основании решения задачи взаимного исключения.

Имеются различные варианты взаимодействия между процессами–писателями и процессами–читателями. Наиболее широко распространены следующие:

- устанавливается высшая приоритетность в использовании критического ресурса процессам–читателям (если хотя бы один процесс–читатель пользуется ресурсом, то он закрыт для использования всеми процессами–писателями);

- наивысший приоритет у процессов–писателей (при появлении запроса от процесса–писателя необходимо закрыть ресурс для использования процессами–читателями).

1.4. Задача «обедающие философы»

Название и формулировка этой задачи носят несколько абстрактный характер, но такая задача синхронизации также имеет место при построении систем распределения ресурсов в составе операционной системы. В рамках этой задачи формулируются требования на синхронизацию работы процессов, которые совместно используют пересекающиеся группы ресурсов.

Рассмотрим формулировку задачи «обедающие философы» в терминологии, предложенной Э. Дейкстрой (Edsser Wybe Dijkstra).

За круглым столом расставлены пять стульев, на каждом из которых сидит определенный философ, $Ф_i$ (рис. 4.1).

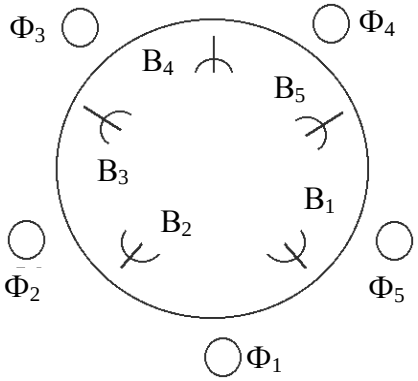


Рис. 4.1. Задача «обедающие философы»

В центре стола – большое блюдо спагетти, а на столе лежат пять вилок В1…В5 – каждая между двумя соседними тарелками.

Любой философ может находиться только в двух состояниях – либо он размышляет, либо ест спагетти. Начать думать философу ничто не мешает. Но чтобы начать есть, философу нужны две вилки: одна в правой руке, другая в левой. Закончив еду, философ кладет вилки слева и справа от своей тарелки и опять начинает размышлять до тех пор, пока снова не проголодается.

Существует множество различных формулировок данной задачи, в одной из которых философы интерпретируются как процессы, а вилки – как ресурсы.

В представленной задаче имеются две опасные ситуации: ситуация голодной смерти и ситуации голодания отдельного философа.

Ситуация голодной смерти возникает в случае, когда философы одновременно проголодаются и одновременно попытаются взять, например, свою левую вилку. В данном случае возникает тупиковая ситуация, так как никто из них не может начать есть, не имея второй вилки. Для исключения ситуации голодной смерти были предложены различные стратегии распределения вилок.

Ситуация голодания возникает в случае заговора двух соседей слева и справа против философа, в отношении которого строятся козни. Заговорщики поочередно забирают вилки то слева, то справа от него. Такие согласованные действия злоумышленников приводят жертву к вынужденному голоданию, так как он никогда не может воспользоваться обеими вилками.

В рассмотренной системе тупиков можно избежать, используя ограничение на пользование вилками, согласно которому философ может взять две вилки, необходимые для еды, только в том случае, если обе они свободны (предполагается, что каждый из философов берет две необходимые для еды вилки одновременно). Фаза ожидания каждого философа должна быть конечной (следовательно, система будет свободна от ситуации голодания).

Другим способом избегания тупиков является введение еще одной задачи, гарантирующей, что в каждый данный момент времени за столом находится не более четырех философов. Каждый философ прежде, чем сесть за стол, должен получить разрешение задачи, а покидая стол, уведомить задачу об этом. Тупиков быть не может, так как за столом будет, по крайней мере, один философ, который может приступить к еде. После еды, занимающей конечное время, он уйдет, и другой философ сможет начать прием пищи.

Примечания.

1. Задача «обедающие философы» удобна для изучения тупиковых ситуаций в системах параллельных процессов.

2. При решении рассмотренных выше задач можно применять переменные состояния, которые анализируются при вхождении процесса в критическую область, семафорные механизмы (простые, множественные, счетные семафоры), мониторoподобные механизмы синхронизации.

2. Ход работы

2.1. Изучите теоретические сведения.

2.2. Рассмотрите примеры:

Пример 1

Тема: синхронизация работы процессов, совместно использующих пересекающиеся группы ресурсов.

Дано:

- имеются три параллельных процесса X, Y и Z и три ресурса P_1, P_2 и P_3;
- для пребывания процесса X в активном состоянии ему требуется выделить одновременно ресурсы P_1 и P_2;
- для пребывания процесса Y в активном состоянии – ресурсы P_2 и P_3;
- для пребывания процесса Z в активном состоянии – ресурсы P_3 и P_1;
- скорости развития процессов произвольные;
- переходы из активного в другие состояния происходят в непредсказуемые моменты.

Необходимо обеспечить максимальное параллельное и правильное развитие процессов.

Решение проблемы: синхронизация в данном случае заключается в определенном упорядочении действий процессов по захвату ресурсов во избежание возможных блокировок одними процессами других.

В данном случае возможны две опасности:

- возникновение тупиковой ситуации при распределении ресурсов;
- возникновение ситуации «голодания» в отношении некоторого процесса при распределении ресурса.

2.3. Выполните задания.

1. Разработайте сеть Петри, которая моделирует механизм взаимного исключения для процессов P_1 и P_2.

Дано: два процесса P_1 и P_2 работают с разделяемым ресурсом, используя значение x из данного разделяемого объекта. При этом процесс P_1 вычисляет значение $x' = f(x)$, а процесс P_2 – значение $x'' = g(x)$, оба процесса заносят полученные значения в разделяемый объект.

П р и м е ч а н и е. Если процессы, при отсутствии синхронизации, в одно и то же время пытаются выполнить требуемые действия, то содержимое разделяемого ресурса может быть непредсказуемым. Так, если процессы P_1 и P_2 в одно и то же время пытаются выполнить такую последовательность действий, то могут возникнуть искажения данных. Например, возможна следующая последовательность:

1) процесс P_1 считывает значение x из разделяемого объекта;
2) процесс P_2 считывает значение x из разделяемого объекта;
3) процесс P_1 вычисляет новое значение $x' = f(x)$;
4) процесс P_2 вычисляет новое значение $x'' = g(x)$;
5) процесс P_1 записывает x' в разделяемый объект;
6) процесс P_2 записывает x'' в разделяемый объект, уничтожая значение x.

Результат вычисления процесса P_1 потерян.

2. Промоделируйте сетью Петри взаимодействие процесса–производителя и процесса–потребителя.

Дано:

- буфер – разделяемый объект, посредством которого реализуется взаимодействие через асинхронную передачу сообщений;
- система из двух параллельных процессов, включающая процесс–производитель и процесс–потребитель;
- процесс–производитель создает сообщения, которые помещаются в буфер;
- процесс–потребитель ждет, пока сообщение не будет помещено в буфер, извлекает его оттуда и использует.

П р и м е ч а н и е. Каждая фишка должна соответствовать сообщению, которое произведено, но еще не использовано. Следует использовать дополнительную позицию, представляющую буфер.

3. Промоделируйте сетью Петри взаимодействие процессов, отражающих классическую задачу «обедающие философы» (условие на с. 29–31 данного пособия).

Контрольные вопросы

1. В чем суть концепции параллелизма вычислительных процессов?
2. Какими аппаратными средствами обеспечивается эффективное использование внутреннего параллелизма задачи?
3. Какие отношения между процессами определяют правила синхронизации?
4. Задача «взаимного исключения», ее сущность.
5. Задача «производители–потребители», ее сущность.
6. Задача «читатели–писатели», ее сущность.
7. Задача «обедающие философы», ее сущность.
8. Какие средства синхронизации предпочтительнее использовать при решении каждой из вышеуказанных задач?

Содержание отчета

1. Цель работы.
2. Краткое описание позиций и переходов разрабатываемых моделей.
3. Модели взаимодействующих параллельных процессов в виде сетей Петри.
4. Выводы по работе.

ЛАБОРАТОРНО-ПРАКТИЧЕСКАЯ РАБОТА 20

МОДЕЛИРОВАНИЕ ВЗАИМОДЕЙСТВИЯ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Цель работы: приобретение навыков моделирования взаимодействия параллельных процессов с использованием языков программирования.

Техническое обеспечение: ПЭВМ на базе Intel-80486.

Программное обеспечение: OS Windows XP.

Время выполнения: 4 часа.

1. Краткие теоретические сведения

Использование объектов синхронизации. При многопоточной обработке могут возникнуть следующие ситуации:

- рассинхронизация (*race conditions*);
- тупиковая ситуация (*deadlock* – «смертельное объятие»).

Пример 1

Тема: Race conditions.

Рассмотрим ситуацию, когда успех одной операции зависит от успеха другой, но обе они не синхронизированы друг с другом.

синхронизированы и Поток1 не успеет выполнить свою работу до того, как начнется печать, то получим сбой.

Из истории вопроса. Голландский профессор математики Э. Дейкстра в начале 70-х годов XX века, рассматривая ситуации с многопользовательским доступом, ввел следующие понятия (термины взяты из учебника информатики Мюнхенского технологического университета (1973 год)):

- критический интервал (*critical section*);
- семафоры (*semaphore*);
- взаимные исключения (*mutex*).

Под *критическим интервалом* он определил всякую очередность шагов обработки, в которой последовательный процесс не должен прерываться никаким другим процессом. Он сопоставил каждому, общему для некоторых процессов, набору данных запирающую переменную (*mutex* – mutual exclusion), начальное значение которой равно 1.

Если к началу критического интервала она больше или равна 1, то интервал должен выполняться, иначе – «ждать».

Алгоритм:

```
Mutex=1;
В начале критического интервала:
    mutex=mutex – 1;
    if (mutex<0)
    {
    //ждать;
    } else {
    //выполнять;
    };
В конце критического интервала:
    mutex=mutex+1;
    if (mutex<=0)
    {
    //взять из очереди следующий процесс и выполнять его
    } else {
    //процессов больше нет
    };
```

Во время критического интервала значение *mutex* равно нулю или отрицательному числу. Абсолютное значение его равно длине очереди.

Вследствие ошибок при программировании синхронных событий могут возникать системные заторы (*system-deadlock*) или мертвые хватки (*deadly embrace*).

Ожидающие функции ряда Wait. Эти функции блокируют выполнение потока до наступления какого-то события или тайм-аута.

Для того чтобы пользоваться этими функциями, должен быть объект синхронизации, который проверяет эти функции.

У этих объектов два состояния: «установлен» и «сброшен».

Алгоритм использования ожидающих функций:

- вызов функций (им передается указатель на объект синхронизации);
- объект проверяется;
- если объект не установлен, то функция будет ждать, пока не истечет тайм-аут, все это время поток будет блокирован.

Сценарий синхронизации потоков с использованием ожидающих функций:

- прежде чем заснуть, поток сообщает системе то особое событие, которое должно разбудить его;
- при засыпании потока операционная система перестает выделять ему кванты процессорного времени, приостанавливая его выполнение;
- как только указанное потоком событие произойдет, система возобновит выдачу ему квантов процессорного времени и поток вновь может развиваться.

1.1. Объект Семафор (Semaphore)

Объектами синхронизации называются объекты Windows, идентификаторы которых могут использоваться в функциях синхронизации. Среди них есть как объекты, использующиеся только для синхронизации, так и объекты, которые применяются в других целях, но могут вызывать срабатывание функций ожидания. Рассмотрим объект **Семафор**, использующийся только для синхронизации.

Семафором называется объект ядра, который позволяет только одному процессу или одному потоку процесса обратиться к критической секции – блоку кодов, осуществляющему доступ к объекту.

Серверы баз данных используют их для защиты разделяемых данных.

Классический семафор был создан Э. Дейкстрой, который описал его в виде объекта, обеспечивающего выполнение двух операций *P* и *V*:

P – сокращение голландского слова Proberen, что означает тестирование;

V – сокращение глагола Verhogen, что означает приращивать (increment).

Различают два основных использования семафоров: защита критической секции и обеспечение совместного доступа к ресурсу.

Примеры использования семафоров:

- критическая секция в виде функции, осуществляющей доступ к таблице базы данных;
- реализация списка операционной системы, который называется *process control blocks* (PCBs). Это список указателей на активные процессы. В каждый конкретный момент времени только один поток ядра системы может изменять этот список, иначе будет нарушена семантика его использования;
- управление перемещением данных (*data flow*) между *n* производителями и *m* потребителями. Существует много систем, имеющих архитектуру типа *data flow*. В них выход одного блока функциональной схемы целиком поступает на вход другого блока. Когда потребители хотят получить данные, они выполняют операцию типа *P*. Когда производители создают данные, они выполняют операцию типа *V*.

Изначально семафоры создавались как глобальные структуры, совместно с глобальными API-функциями, реализующими операции *P* и *V*.

Теперь семафоры реализуются в виде класса объектов. Обычно абстрактный класс **Семафор** определяет чисто виртуальные функции типа *P* и *V*. От него производятся классы, реализующие два указанных типа семафоров: защита критической секции и обеспечение совместного доступа к ресурсу. В смысле видимости оба типа семафоров могут быть объявлены либо глобально во всей операционной системе, либо глобально в пространстве процесса. Первые видны всем процессам системы и, следовательно, могут ими управлять. Вторые действуют только в пространстве одного процесса и, следовательно, могут управлять его потоками.

Сам семафор ничего не знает о том, что он защищает. Ему обычно передается ссылка на объект класса, который хочет использовать критическую секцию, и он либо дает доступ к объекту, либо приостанавливает («усыпляет» – *suspends)* объект до тех пор, пока доступ не станет возможным. Важно отметить, что при реализации семафоров и других объектов ядра используют специальные атомарные команды (*atomic steps*), которые не прерываются системой.

Семафор позволяет ограничить доступ потоков к объекту синхронизации на основании их количества.

Пример алгоритма работы с семафором. Требуется, чтобы к какому-нибудь объекту могли обратиться максимум 3 потока:

- семафор инициализируется, ему передается количество потоков, которые к нему могут обратиться;

- при каждом обращении к ресурсу счетчик семафора уменьшается. Когда счетчик уменьшится до 0, к ресурсу обратиться больше нельзя;

- при отсоединении потока от семафора его счетчик увеличивается, что позволяет другим потокам обратиться к нему;

- сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

1.2. Объект Критическая секция (Critical section)

Критическая секция (**Critical section**) – это участок кода, в котором поток (*thread*) получает доступ к ресурсу (например, переменной), доступному из других потоков.

Этим объектом может владеть только один поток, что и обеспечивает синхронизацию.

Если поток обратится к критической секции, в которой сейчас другой поток, то он будет блокирован, пока критическая секция не будет освобождена.

П р и м е ч а н и е. Используя механизм синхронизации, надо помнить о том, что физически данные не защищены: никто не помешает потоку обратиться к разделяемым данным. Синхронизация – это подсказка, как должен себя вести объект при доступе к данным.

1.3. Объект Мьютекс (Mutex)

Критические секции просты в использовании и обладают высоким быстродействием, но не обладают гибкостью в управлении.

При их использовании нет возможности:

- установить время блокирования;

- присвоить имя критической секции для того, чтобы два разных процесса могли ее использовать.

Оба эти недостатка можно устранить, если использовать такой объект ядра, как **Mutex**. Этот объект применяется, например, для синхронизации доступа к файлам.

Общая идея взаимоисключения похожа на критические секции. Только **Mutex** позволяет проводить синхронизацию как между потоками (*thread*), так и процессами (*process*), то есть между приложениями.

Термин «mutex» происходит от *mutually exclusive* (взаимно исключающий), так как этот объект обеспечивает исключительный (*exclusive*) доступ к охраняемому блоку кодов.

Данный объект синхронизации регистрирует доступ к ресурсам и может находиться в двух состояниях:

- установлен;

- сброшен.

Mutex – установлен в тот момент, когда ресурс свободен.

Пример 2

***Тема*:** использование объекта **Mutex**.

Дано: несколько процессов должны одновременно работать с одним и тем же связным списком.

Требуется: на время выполнения каждой операции (добавления, удаления элемента или сортировки) заблокировать список и разрешить доступ к нему только одному из процессов.

Решение:

- если в данный момент объект не занят, то **Mutex** предоставляет доступ к объекту любому из потоков, принадлежащих разным процессам, и запоминает текущее состояние объекта;

- если объект занят, то **Mutex** запрещает доступ.

1.4. Объект Событие (Event)

В некоторых случаях потоку необходимо ждать, пока другие потоки не завершат выполнение каких-то операций или не произойдет какое-либо событие (*UI*-событие *User Interface*), то есть событие, инициированное пользователем.

Пусть поток, копирующий данные в архив, должен быть уведомлен о том, что поступили новые данные. Использование объекта **Event** позволяет оптимально справиться с этой задачей.

Смысл использования события – в уведомлении одного или нескольких ожидающих потоков. Объект **Event** может находиться в двух состояниях – занят и свободен.

События (**Event**) бывают двух типов:

1) сброс вручную;

2) сбрасываются ожидаемыми функциями.

П р и м е ч а н и е. Первый вид события нужно применять, если событие ждут несколько потоков. Только сброс вручную позволяет это сделать. Иначе первый же обработчик сбросит событие, и другие потоки об этом не узнают.

2. Ход работы

2.1. Изучите теоретические сведения.

2.2. Рассмотрите примеры:

Пример 1

Тема: использование счетного семафора *S* для решения задачи типа «читатели–писатели».

Дано:

- работа с некоторой файловой системой;

- число процессов–писателей и процессов–читателей произвольно.

П р и м е ч а н и я.

1. С файловой системой одновременно могут работать (читать информацию) сколько угодно процессов–читателей.

2. Одновременная работа по записи и считыванию информации недопустима.

3. С некоторым файлом одновременно не могут работать процесс–писатель и процесс–читатель.

4. Недопустима одновременная работа процессов–писателей с файлами.

```
VAR S: semaphore;
Q, R: integer;
  BEGIN R:=1; Q:=n; {инициализация}
    COBEGIN
          Потребитель 1: REPEAT
                <произвольные операторы>;
                P(S,R); <чтение из базы данных>;
                V(S,R);
             FOREVER;
          ..................
          Потребитель k: REPEAT
                <произвольные операторы>;
                P(S,R); <чтение из базы данных>;
                V(S,R);
            FOREVER;
          Производитель 1: REPEAT
                <произвольные операторы>;
                P(S,Q); <запись в базу данных>;
                V(S,Q);
            FOREVER;
          ..................
          Производитель k: REPEAT
                <произвольные операторы>;
                P(S,Q); <запись в базу данных>;
                V(S,Q);
                FOREVER
    COEND
  END
```

Пример 2

Тема: использование множественных семафоров для решения задачи «обедающие философы».

Дано:

- каждый философ – это отдельный процесс с номером I=1...5;

- зная свой номер, философ (процесс) в состоянии определить номер соседа слева. LEFT:=(I–1) MOD 5 и номер соседа справа RIGTH:=(I+1) MOD 5;

- каждой вилке поставлен в соответствие отдельный семафор;

- все семафоры составляют отдельный массив FORKS;

- если I вилка в текущий момент времени занята философом, то значение семафора равно 0;

- каждый примитив *P*- и *V*- проводит обработку сразу двух семафоров из массива;

- семафоры неделимы и исключают друг друга во времени.

```
VAR FORKS: ARRAY 1...5 of semaphore; I: integer;
  BEGIN
    I=:5;
    REPEAT FORKS [I]:=1; I:=I–1 UNTIL I=0;
    COBEGIN
        1: BEGIN ... <тело процесса> ... END;
        ......
        I: BEGIN
            VAR LEFT, RIGTH:1...5;
            BEGIN
                LEFT:=(I-1) MOD 5;
                RIGTH:=(I+1) MOD 5;
                REPEAT
                <размышления>;
                P(FORKS[LEFT]; FORKS[RIGTH]);
                        <использование ресурса>;
                        V(FORKS[LEFT]; FORKS[RIGTH])
                FOREVER
            END
        END
    COEND
  END
```

При выполнении примитива *P*- процесс может заснуть:

- на одном семафоре;
- на другом;
- на двух сразу.

Такие блокировки развития процесса возникают, если в момент выполнения примитива *P*- соответствующие проверяемые семафоры равны 0. В предложенном решении исключено активное ожидание, которое пришлось бы реализовать, если для решения данной задачи использовались бы не множественные, а простые семафоры.

2.3. Выполните задания.

1. Смоделируйте работу магазина, используя многопоточность.

Дано:

a) покупатели:

1) ждут входа в магазин;
2) делают случайное количество покупок;
3) случайное время проводят в кафе;
4) становятся в очередь у кассы;
5) проводят у кассы случайный промежуток времени;
6) отходят от кассы;
7) уходят из магазина;

b) магазин:

1) открытие магазина;
2) создание потоков моделирующих отдельных покупателей;
3) закрытие входа по окончании работы (после этого в магазин не может войти ни один покупатель);
4) ожидание оставшихся в магазине покупателей, которые постепенно расплачиваются и покидают магазин.

Магазин характеризуется следующими параметрами:

- максимальным количеством покупателей;
- временем работы в абстрактных единицах;
- количеством касс;
- промежутком времени между созданием покупателей.

Покупатели характеризуются следующими параметрами:

- временем ожидания у входа в магазин;
- временем покупок;
- временем обслуживания у кассы;
- временем пребывания в кафе;
- временем ожидания в очереди и кафе.

Примечания.

1. Каждый покупатель – отдельный поток.
2. Магазин – отдельный поток.

2. Разработайте программу решения задачи «читатели-писатели».

3. Разработайте программу решения задачи «производители-потребители».

4. Разработайте программу решения задачи «обедающие философы».

Требуется реализовать приложение, отображающее жизнедеятельность философов таким образом, чтобы тупиковые ситуации никогда не наступили.

Дано:

1) за круглым столом расставлены пять стульев;
2) на каждом из стульев сидит определенный философ ($Ф_i$);
3) в центре стола – большое блюдо спагетти, а на столе лежат пять вилок (В1…В5), каждая из которых – между двумя соседними тарелками;
4) любой философ может находиться только в двух состояниях:

- либо он размышляет;
- либо ест спагетти;

5) начать думать философу ничто не мешает;
6) чтобы начать есть, философу нужны две вилки: одна в правой руке, другая – в левой;
7) закончив еду, философ кладет вилки слева и справа от своей тарелки и опять начинает размышлять до тех пор, пока снова не проголодается.

Примечание. Философы интерпретируются как процессы, а вилки – как ресурсы.

В представленной задаче имеются две опасные ситуации:

- ситуация голодной смерти;
- ситуации голодания отдельного философа.

Моделирование развития параллельных процессов можно производить на основе абстрактных процессов, которым назначают имена, приоритеты, время обслуживания запросов (в условных единицах), количество требуемого абстрактного ресурса (в условных единицах). Для построения программы необходимо использовать механизмы синхронизации параллельных процес-

сов, изученные в теоретическом курсе. Выбор и реализация механизмов может быть произвольной. Программа модели развития параллельных процессов и методов их синхронизации должна наглядно представлять порядок использования процессами критических ресурсов.

Входными данными программы должны быть: список имен процессов с указанием их приоритетов и требуемого времени обслуживания запроса (в условных единицах). Необходимо предусмотреть возможность динамического изменения заданных параметров. При выполнении программы на экране должно быть наглядное отображение ситуации, складывающейся при взаимодействии процессов, состояние используемых переменных (семафоров). При реализации программ моделирования взаимодействия параллельных процессов можно использовать организацию очередей.

Примечание. Для реализации многопоточной обработки в Windows XP воспользуйтесь сведениями, приведенными в приложении 1.

Контрольные вопросы

1. Что такое «ожидающие» функции?
2. Что представляет собой объект **Semaphore** и каковы особенности его использования?
3. В чем заключается сущность понятия «критическая секция»?
4. Что представляет собой объект **Mutex**?
5. Каковы особенности использования объекта **Event**?

Содержание отчета

1. Цель работы.
2. Краткое описание используемых механизмов синхронизации и обоснование их выбора для реализации задач.
3. Реализация задач на алгоритмическом языке.
4. Выводы по работе.

ЛАБОРАТОРНО-ПРАКТИЧЕСКАЯ РАБОТА 21

МОДЕЛИРОВАНИЕ АЛГОРИТМОВ ПРЕДОТВРАЩЕНИЯ И ОБХОДА ТУПИКОВЫХ СИТУАЦИЙ

Цель работы: приобретение навыков моделирования алгоритмов предотвращения и обхода тупиковых ситуаций при взаимодействии параллельных процессов.

Техническое обеспечение: ПЭВМ на базе Intel-80486.
Программное обеспечение: OS Windows XP.
Время выполнения: 4 часа.

1. Краткие теоретические сведения

1.1. Понятие тупиковой ситуации. Условия возникновения deadlock

При параллельном исполнении процессов могут возникнуть такие ситуации, при которых два или более процессов все время находятся в заблокированном состоянии.

Для возникновения тупиковой ситуации необходимо, чтобы одновременно выполнялись четыре условия:

1) ***взаимного исключения***, при котором процессы осуществляют монопольный доступ к ресурсам;

2) ***ожидания***, при котором процесс, запросивший ресурс, ждет до тех пор, пока запрос не будет удовлетворен, при этом удерживая ранее полученные ресурсы;

3) ***отсутствия перераспределения***, при котором ресурсы нельзя отбирать у процесса, если они уже ему выделены;

4) ***кругового ожидания***, при котором существует замкнутая цепь процессов, каждый из которых ждет ресурс, удерживаемый его предшественником в этой цепи.

Для решения проблемы тупиковой ситуации, можно выбрать одну из трех стратегий:

1) стратегия ***предотвращения deadlock*** (запрет существования опасных состояний) – тупиковые ситуации настолько дорогостоящи, что лучше потратить дополнительные ресурсы системы для сведения к нулю вероятности возникновения тупиковых ситуаций при любых обстоятельствах;

2) стратегия ***обхода deadlock*** (запрет входа в опасное состояние) гарантирует, что тупиковая ситуация, хотя в принципе и возможна, но не возникает для конкретного набора процессов и запросов, выполняющихся в данный момент;

3) стратегия ***распознавания deadlock и последующего восстановления*** (запрет постоянного пребывания в опасном состоянии) базируется на том, что тупиковая ситуация возникает достаточно редко, и поэтому предпочтительнее просто распознать ее и произвести восстановление, чем применять стратегии предотвращения или обхода тупика.

Дисциплина, предотвращающая *deadlock*, должна гарантировать, что хотя бы одно из четырех условий, необходимых для его возникновения, не наступит.

Поэтому для предотвращения *deadlock* следует подавить хотя бы одно из следующих условий:

- *условие взаимного исключения* подавляется путем разрешения неограниченного разделения ресурсов;

- *условие ожидания* подавляется предварительным выделением ресурсов. Процесс может потребовать все ресурсы заранее, и он не может начать выполнение до тех пор, пока они ему не будут выделены. Следовательно, общее число ресурсов, необходимое параллельным процессам, не может превышать возможности системы. Каждый процесс должен ждать до тех пор, пока не получит все необходимые ресурсы, даже если один из них используется только в конце исполнения процесса;

- *условие отсутствия перераспределения* подавляется разрешением операционной системе отнимать у процесса ресурсы. Это возможно, если можно запомнить состояние процесса для его последующего восстановления;

- *условие кругового ожидания* подавляется предотвращением образования цепи запросов, что можно обеспечить иерархическим выделением ресурсов. Все ресурсы образуют некоторую иерархию. Процесс, затребовавший ресурс на одном уровне, может затем потребовать ресурсы на более высоком уровне. Он может освободить ресурсы на данном уровне только после освобождения всех ресурсов на всех более высоких уровнях.

Когда последовательность запросов, связанных с каждым процессом, неизвестна заранее, но известен общий запрос на ресурсы каждого типа, то выделение ресурсов можно контролировать: для каждого требования, предполагая, что оно удовлетворено, надо определять, существует ли среди общих запросов некоторая последовательность требований, которая может привести к опасному состоянию.

Вход в опасное состояние можно предотвратить, если у системы есть информация о последовательности запросов, связанных с каждым параллельным процессом.

Если вычисления находятся в любом неопасном состоянии, то существует, по крайней мере, одна последовательность состояний, которая обходит опасное состояние. Следовательно, достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, запрос отклоняется. Если нет, его можно выполнить. Проверка того, является состояние опасным или нет, требует анализа последующих запросов процессов. Существуют методы эффективного выполнения такого просмотра. Данный подход является примером контролируемого выделения ресурса. Классическое решение этой задачи известно как *«алгоритм банкира»*.

1.2. «Алгоритм банкира»

Банкир ссужает денежные суммы (в одной валюте) некоторому числу клиентов, каждый из которых заранее сообщает банкиру максимальную сумму, которая ему будет нужна. Клиент может занимать эту сумму по частям, и нет никакой гарантии, что он возвратит часть денег до того, как сделает весь заем. Весь капитал банкира обычно меньше, чем суммарные требования клиентов, так как банкир не предполагает, что все в некоторый момент сделают максимальные заемы одновременно. Если в некоторый момент клиент запросит денежную сумму, банкир должен знать, сможет ли он ссудить ее без риска попасть в ситуацию, когда не будет достаточного количества денег, чтобы обеспечить дальнейшие заемы, а именно это в конце концов позволяет клиентам возвратить долг. Чтобы решить проблему:

- банкир предполагает, что выполнил запрос и оценивает сложившуюся ситуацию;

- определяет клиента, чей текущий заем наиболее близок к максимальному;

- если банкир не может ссудить оставшуюся сумму этому клиенту, то он отклоняет первоначальный запрос;

- если же банкир может ссудить оставшуюся сумму, то он предполагает, что этот клиент полностью рассчитался, и обращает свое внимание на того клиента из оставшихся, чей запрос ближе всего к своему лимиту;

- просматривая, таким образом, всех клиентов, банкир каждый раз проверяет, будет ли достаточно денег, чтобы удовлетворить минимальный заем клиента и предоставить последнему полную сумму. В случае если это так, банкир удовлетворяет первоначальный заем.

2. Ход работы

2.1. Изучите теоретические сведения.

2.2. Рассмотрите примеры:

Пример 1

Тема: Deadlock – каждый поток ждет событие, которое никогда не наступит.

Дано: пусть один поток ждет завершения второго, а второй – завершения первого события.

Поток1 реализует следующие функции:

- блокирует запись, идентифицирующую клиента;
- блокирует запись, описывающую его счет;
- изменяет обе записи;
- освобождает запись, описывающую счет;
- освобождает запись, идентифицирующую клиента.

Таким образом, освобождение блокировок происходит в обратном порядке (именно так следует поступать при работе с записями базы данных и всеми объектами ядра Windows).

Поток2 реализует функцию начисления месячного процента и он делает те же действия, что и первый, но порядок блокирования и освобождения записей обратный.

Оба потока по отдельности функционируют нормально. Однако, в процессе работы возможен следующий сценарий:

- Поток1 блокирует запись, идентифицирующую клиента;
- Поток2 блокирует запись, описывающую его счет.

Оба ждут освобождения записей, блокированных друг другом. Если ожидание реализовано в виде бесконечного цикла, то бесконечный цикл получен. Это тупиковая ситуация, или *deadlock*.

Пример 2

Тема: синхронизация нескольких потоков, при условии, что объектов синхронизации несколько.

Дано: два потока должны в ходе своей работы захватывать два ресурса: монитор и клавиатуру (только в том случае, когда оба эти ресурса захвачены, поток может продолжаться).

Возможная проблема: два потока ждут друг друга, и это ожидание может продолжаться сколько угодно. Появление данной ситуации нерегулярно, как аппаратный сбой, и это связано с квантованием времени.

Рассмотрим пример:

1) операционная система выделила время Потоку1;
2) Поток1 захватил монитор;
3) Поток1 захватил клавиатуру;
4) операционная система выделила время Потоку2;
5) Поток2 сделал попытку захватить монитор, но безуспешно – занято;
6) Поток2 находится в ожидании;
7) операционная система выделила время Потоку1;
8) Поток1 выполнил какие-то действия и освободил монитор;
9) операционная система выделила время Потоку2;
10) Поток2 захватил монитор и ждет клавиатуру.

Все работает нормально.

Если бы в первой ситуации за один квант времени Поток1 не успел бы захватить оба объекта, а захватил бы только один, произошла бы тупиковая ситуация.

Для реализации тупиковой ситуации нужно, чтобы два потока в одно и то же время пытались захватить ресурсы. Кроме того, кванта времени, который был выделен для потока, должно не хватить для захвата всех ресурсов.

Решение проблемы: если захват полностью не удался, то есть все необходимые ресурсы не захвачены, надо освободить все и попробовать вновь через некоторое время.

Пример 3

Тема: обход тупика.

Дано:

- три процесса претендуют на ресурс, выделяемый дискретными взаимозаменяемыми единицами;
- пусть в системе всего имеется 10 единиц ресурса;
- процесс А, запросивший 3 единицы, уже имеет 2 единицы, а его максимальная потребность в ресурсе – 6 единиц. Процесс В, запросивший 2 единицы, уже имеет 3 единицы, а его максимальная потребность в ресурсе – 7 единиц. Процесс С, запросивший 3 единицы, уже имеет 2 единицы, а его максимальная потребность в ресурсе – 5 единиц.

Если запрос процесса С будет удовлетворен первым, то процесс сможет продолжить исполнение и в конце концов освободит все 5 единиц, которые у него были. Это позволит процессу В, а затем процессу А получить все необходимые им ресурсы и завершиться.

Если же вначале будет выполнен запрос процесса B, то ни один из процессов не сможет завершиться, и в результате, возникнет deadlock, причиной которого являются неупорядоченные попытки процессов войти в критическую область, связанную с выделением соответствующей единицы ресурса.

2.3. Выполните задание.

Разработайте программу реализации «алгоритма банкира» при распределении структурированного ресурса между параллельными процессами.

Входными данными программы являются имена, приоритеты абстрактных процессов, заданное количество структурированного ресурса, необходимое количество ресурса каждому процессу.

Выбор средств для реализации данного алгоритма произвольный: можно использовать семафорные, мониторoподобные механизмы и их сочетания. При выполнении программы должна быть наглядно отображена невозможность возникновения *deadlock* или возможность его обхода.

П р и м е ч а н и е. Для реализации этой задачи в Windows XP воспользуйтесь сведениями, приведенными в приложении 1.

Контрольные вопросы

1. Какие вы знаете условия возникновения *deadlock*?
2. Каковы стратегии решения тупиковых ситуаций?
3. На чем базируется дисциплина предотвращения тупика?
4. В чем заключается сущность «алгоритма банкира»?

Контрольный тест

1. Тупик – это:

- бесконечное ожидание из-за непродуманных алгоритмов блокировки и активизации процессов;
- взаимные блокировки процессов при ожидании ресурсов;
- состояние ожидания события, которое никогда не произойдет;
- явление, имеющее место, когда один процесс приостановил другой процесс, владеющий ресурсом, а сам перешел к ожиданию этого ресурса.

2. О потенциальной опасности тупика свидетельствует следующая характеристика графа Ресурсы-Процессы:

- наличие цепей в графе;
- наличие простых цепей в графе;
- наличие циклических маршрутов в графе;
- наличие кратчайшего пути в графе;
- наличие в графе пути с минимальным весом.

3. Необходимые условия возникновения тупика:

- взаимное исключение при доступе к ресурсу;
- ожидание ресурса без освобождения занятого ресурса;
- динамическое распределение ресурсов между процессами;
- статическое распределение ресурсов между процессами;
- неперераспределяемость ресурсов;
- количество ресурсов, запрашиваемых всеми процессами, больше имеющегося количества ресурсов;
- круговое ожидание ресурсов;
- количество имеющихся ресурсов изменяется со временем;
- при выполнении процессы меняют требования к количеству запрашиваемых ресурсов.

4. Суть метода глобального предотвращения тупиков заключается в следующем:

- процесс запрашивает ресурсы в количестве, существенно превышающем необходимое;
- процесс не запускается, пока запрошенные ресурсы не будут иметься в наличии;
- процесс приостанавливает все другие процессы, запрашивающие те же ресурсы;
- процесс отбирает требуемые ресурсы у других процессов, владеющих ими;
- процесс обращается за поиском требуемых ресурсов в Интернет.

5. Суть нарушения условия неперераспределяемости ресурсов состоит в следующем:

- процесс отдает захваченные ресурсы по первому требованию других процессов;
- процесс отдает захваченные ресурсы при переходе к ожиданию других ресурсов;
- процесс отдает захваченные ресурсы по запросу операционной системы;

- процесс отдает захваченные ресурсы по истечении заданного интервала времени;

- процесс отдает захваченные ресурсы при возникновении заранее установленного события.

6. Суть метода упорядоченных ресурсов состоит в следующем:

- процесс может получить ресурс из класса с данным номером, если он освободил ресурс из класса с меньшим номером;

- процесс может получить ресурс из класса с данным номером, если он заявил об этом запросе до начала выполнения;

- процесс может получить ресурс из класса с данным номером, если другие ресурсы этого класса не захвачены другими процессами;

- процесс не может получить ресурс из класса с данным номером, если он уже владеет ресурсами из данного класса;

- процесс не может получить ресурс из класса с данным номером, если этот класс не принадлежит к числу заранее зарезервированных классов.

Содержание отчета

1. Цель работы.
2. Краткое описание используемого механизма обхода *deadlock*.
3. Реализация задачи на алгоритмическом языке.
4. Выводы по работе.

Рекомендуемая литература

Основная

1. Вебер, Дж. Технология Win32 в примерах / Дж. Вебер. – СПб. : БХВ, 1998. – 1070 с.
2. Воеводин, В. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. – СПб. : БХВ-Петербург, 2002. – 608 с. : ил.
3. Гордеев, А. В. Системное программное обеспечение : учебник / А. В. Гордеев, А. Ю. Молчанов. – СПб. : Питер, 2002. – 736 с. : ил.
4. Клочко, В. И. Теория вычислительных процессов и структур : учеб. пособие / В. И. Клочко. – Краснодар : КубГГУ, 1999.
5. Концептуальное моделирование информационных систем / под ред. В. В. Фильчакова. – СПб. : СПВУРЭ ПВО, 1998.
6. Немнюгин, С. А. Параллельное программирование для многопроцессорных вычислительных систем / С. А. Немнюгин, О. Л. Стесик. – СПб. : БХВ-Петербург, 2002. – 400 с. : ил.
7. Олифер, Н. А. Сетевые операционные системы / Н. А. Олифер, В. Г. Олифер. – СПб. : Питер, 2001.
8. Рихтер, Дж. Windows для профессионалов : создание эффективных Win-32-приложений с учетом специфики 64-разрядной версии Windows : пер. с англ. / Дж. Рихтер. – 4-е изд. – СПб. : Питер ; М. : Русская Редакция, 2004. – 749 с. : ил.
9. Таненбаум, Э. Современные операционные системы : пер. с англ. / Э. Таненбаум. – 2-е изд. – СПб. : Питер, 2007. – 1040 с. : ил.
10. Харт, Джонсон. Системное программирование в среде Win32 : пер. с англ. / Джонсон Харт. – 2-е изд. – М. : Вильямс, 2001. – 464 с. : ил.
11. Хэвиленд, К. Системное программирование в UNIX : руководство программиста по разработке ПО / К. Хэвиленд, Д. Грэй, Б. Салама. – М. : Пресс, 2000.

Дополнительная

12. RED HAT Linux 6.2 / под ред. А. Пасечника. – СПб. : Питер, 2000.
13. Windows 2000 для системного администратора. Microsoft Windows 2000 : Server и Professional. Русские версии / под общ. ред. А. Н. Чекмарева, Д. Б. Вишнякова. – СПб. : BHV, 2000.

14. Багласова, Т. Г. Теория вычислительных процессов и структур : лаб. практикум / Т. Г. Багласова. – Минск : МГВРК, 2000. – 20 с.
15. Программирование на параллельных вычислительных системах : пер с англ. / Р. Рэбб [и др.]. – М. : Мир, 1991. – 376 с.
16. Рейчард, К. UNIX : справочник / К. Рейчард, Э. Фосстер-Джонсон. – СПб. : Питер, 2000.
17. Семик, В. П. Технология программирования параллельных вычислительных систем на базе транспьютерных сетей / В. П. Семик, А. Л. Агаронян, М. С. Каменкова // Итоги науки и техники. Техническая кибернетика. – Т. 30. – М. : ВИНИТИ, 1990. – 350 с.
18. Фролов, А. В. Операционная система OS/2 Warp / А. В. Фролов, Г. В. Фролов. – М. : Диалог-МИФИ, 1995.
19. Фролов, А. В. Программирование для Windows NT / А. В. Фролов, Г. В. Фролов. – М. : Диалог-МИФИ, 1996.
20. Шпаковский, Г. И. Организация параллельных ЭВМ и суперскалярных процессоров / Г. И. Шпаковский. – Минск : Белгосуниверситет, 1996.

Приложение 1
(справочное)

РЕАЛИЗАЦИЯ МНОГОПОТОЧНОЙ ОБРАБОТКИ В WINDOWS

1. Потоки, процессы и контексты. Основные понятия

Рассмотрим потоки и средства синхронизации, реализованные в Win32 API.

Системный вызов (*syscall*) – это процесс вызова функции ядра из приложения пользователя.

Режим ядра – код, который выполняется с максимальными привилегиями.

Режим пользователя – код, выполняющийся с пониженными привилегиями. Если этот код будет использовать одну из запрещенных инструкций, сработает *аппаратное исключение*, и пользовательский процесс, чей код исполнял процессор, в большинстве случаев будет прерван.

Поток (*thread*) – это сущность операционной системы, процесс выполнения на процессоре набора инструкций (программного кода). Общее назначение потоков – параллельное выполнение на процессоре двух или более различных задач. Планировщик операционной системы, руководствуясь приоритетом потока, распределяет кванты времени между разными потоками и ставит потоки на выполнение.

Процесс (*process*) – некая абстракция, которая инкапсулирует в себе все ресурсы процесса (открытые файлы, файлы отображенные в память) и их дескрипторы, потоки и т. д. Каждый процесс имеет как минимум один поток. Также каждый процесс имеет свое собственное виртуальное адресное пространство и контекст выполнения, а потоки одного процесса разделяют адресное пространство процесса.

Каждый поток, как и каждый процесс, имеет свой контекст.

Контекст – это структура, в которой сохраняются следующие элементы:

- регистры процессора;
- указатель на стек потока-процесса.

В случае выполнения системного вызова потоком и перехода из режима пользователя в режим ядра происходит смена стека потока на стек ядра.

При переключении выполнения потока одного процесса на поток другого операционная система обновляет некоторые регистры процессора, которые ответственны за механизмы виртуальной памяти, так как разные процессы имеют разное виртуальное адресное пространство.

Рекомендации по использованию потоков:

1) если задача требует интенсивного распараллеливания, надо использовать потоки одного процесса вместо нескольких процессов. Переключение контекста процесса происходит гораздо медленнее, чем контекста потока;

2) при использовании потока не следует злоупотреблять средствами синхронизации, которые требуют системных вызовов ядра, например – мьютексы (*Mutex*), так как переключение в режим ядра – дорогостоящая операция;

3) в случае написания кода, исполняемого с максимальными привилегиями, следует обходиться без использования дополнительных потоков, так как смена контекста потока – дорогостоящая операция.

Особенности реализации потоков в Windows:

- любой поток, созданный в любом процессе, управляется напрямую планировщиком ядра операционной системы. Это классическая модель реализации потоков на уровне ядра – отображение 1:1 потока пользовательского процесса на поток ядра;

- реализована вытесняющая многозадачность (*preemptive multitasking*), то есть поток с более высоким приоритетом может вытеснить текущий поток. Все современные операционные системы используют данный подход;

- все потоки ядра выполняются либо в контексте потока, инициировавшего системный вызов-I/O операцию, либо в контексте потока системного процесса *system*.

1. Работа с потоками

Более одного потока требуется, если приложению надо управлять несколькими действиями, например, одновременным вводом с клавиатуры и с помощью мыши.

Потоки создаются с помощью функции **CreateThread**, куда передается указатель на функцию (функцию потока), которая будет выполняться в созданном потоке.

Функция **CreateThread** возвращает специальное значение типа **handle** потока, который может быть использован для приостановки, уничтожения потока, синхронизации.

Поток считается завершенным, когда выполнится функция потока.

Если же требуется гарантировать завершение потока, то можно воспользоваться функцией **TerminateThread**, которая «убивает» поток, что не всегда корректно.

Функция **ExitThread** будет вызвана неявно, когда ***завершится функция потока***, или же можно вызвать данную функцию самостоятельно. Главная ее задача – освободить стек потока и его **handle**, то есть структуры ядра, которые обслуживают данный поток.

Поток может пребывать в состоянии ***сна*** (**suspend**).

Чтобы «усыпить поток» (приостановить поток извне или из потока), используется функция **SuspendThread** с аргументом, равным **handle** потока.

«Пробуждение» (продолжение выполнения) потока возможно с помощью вызова **ResumeThread**.

Поток можно перевести в состояние сна при создании. Для этого нужно передать в **CreateThread** значение **create_suspended** в предпоследнем аргументе:

CreateThread (0, 0, &threadproc, lparam, create_suspended, &id);

Таким образом, в результате выполнения функции **CreateThread** будет создан новый поток, функция которого начнет выполняться либо сразу же, либо будет приостановлена до вызова **ResumeThread**.

При создании каждому потоку также назначается уникальный **id**.

Повторный вызов **CreateThread** приведет к созданию еще одного потока, выполняющегося одновременно с созданным, и т. д. Таким образом можно создавать неограниченное число потоков, но каждый новый поток ***тормозит*** выполнение остальных.

Для ожидания окончания выполнения потока можно использовать функцию **WaitForSingleObject** с **handle** потока.

3. Механизмы синхронизации операционной системы Windows

В Win32 существуют средства *синхронизации* двух типов:

- реализованные на уровне пользователя (критические секции – **Critical section**);
- реализованные на уровне ядра (мьютексы – **Mutex**, события – **Event**, семафоры – **Semaphore**).

Общие черты механизмов синхронизации:

- используют примитивы ядра при выполнении, что сказывается на производительности;
- могут быть именованными и неименованными;
- работают на уровне системы, то есть могут служить *механизмом межпроцессного взаимодействия* (IPC);
- используют для ожидания и захвата примитива единую функцию: **WaitForSingleObject/WaitForMultipleObjects**.

Существуют несколько *стратегий*, которые могут применяться, чтобы разрешать проблемы, связанные с взаимодействием потоков.

Наиболее распространенным способом является *синхронизация потоков*, суть которой состоит в том, чтобы вынудить один поток ждать, пока другой не закончит какую-то определенную заранее операцию. Для этой цели существуют специальные синхронизирующие объекты ядра операционной системы *Windows*. Они исключают возможность одновременного доступа к тем данным, которые с ними связаны. Их реализация зависит от конкретной ситуации и предпочтений программиста.

Общие положения использования объектов ядра системы:

- однажды созданный объект ядра можно открыть в любом приложении, если оно имеет соответствующие права доступа к нему;
- каждый объект ядра имеет счетчик числа своих пользователей. Как только он станет равным нулю, система уничтожит объект ядра;
- обращаться к объекту ядра надо через ***описатель*** (**handle**), который система дает при создании объекта;
- каждый объект может находиться в одном из двух состояний: свободном (*signaled*) или занятом (*nonsignaled*).

3.1. Работа с объектом Критическая секция (Critical section)

Это самые простые объекты ядра Windows, которые не снижают общей эффективности приложения. Пометив блок кодов в качестве **Critical section**, можно синхронизировать доступ к нему от нескольких потоков.

Для работы с критическими секциями есть ряд функций **API** и тип данных CRITICAL_SECTION. Алгоритм использования следующий:

1) объявить глобальную структуру

CRITICAL_SECTION cs;

2) инициализировать (обычно это делается один раз, перед тем как начнется работа с разделяемым ресурсом) глобальную структуру вызовом функции

InitializeCriticalSection (&cs);

3) поместить охраняемую часть программы внутрь блока, который начинается вызовом функции **EnterCriticalSection** и заканчивается вызовом **LeaveCriticalSection**:

```
EnterCriticalSection (&cs);
{
//====== охраняемый блок кодов
}
LeaveCriticalSection (Scs);
```

Функция **EnterCriticalSection**, анализируя поле структуры **cs**, которое является счетчиком ссылок, выясняет, вызвана ли она в *первый* раз.

Если да, то функция увеличивает значение счетчика и разрешает выполнение потока дальше. При этом выполняется блок, модифицирующий критические данные. Допустим, в это время истекает квант времени, отпущенный данному потоку, или он вытесняется более приоритетным потоком, использующим те же данные.

Новый поток выполняется, пока не встречает функцию **EnterCriticalSection**, которая помнит, что объект **cs** уже занят. Новый поток приостанавливается (засыпает), а остаток процессорного времени передается другому потоку.

Функция **LeaveCriticalSection** уменьшает счетчик ссылок на объект **cs**.

Как только поток покидает критическую секцию, счетчик ссылок обнуляется и система будит ожидающий поток, снимая защиту секции кодов.

Критические секции применяются для синхронизации потоков лишь в пределах одного процесса. Они управляют доступом к данным так, что в каждый конкретный момент времени только один поток может их изменять;

4) когда надобность в синхронизации потоков отпадает, следует вызвать функцию, освобождающую все ресурсы, включенные в критическую секцию:

DeleteCriticalSection (&cs);

П р и м е ч а н и е. Функция **TryEnterCriticalSection()** позволяет проверить критическую секцию на занятость.

Таким образом, поток, который желает обезопасить определенные данные от **race conditions**, вызывает функцию **EnterCriticalSection/TryEnterCriticalSection**:

- если критическая секция свободна, поток занимает ее;
- если же нет, поток блокируется до тех пор, пока секция не будет освобождена другим потоком с помощью вызова функции **LeaveCriticalSection**.

Данные функции – атомарные, то есть целостность данных нарушена не будет.

3.2. Работа с объектом Семафор (Semaphore)

Семафор представляет собой счетчик, содержащий целое число в диапазоне от 0 до максимальной величины, заданной при его создании.

Для работы с объектом **Semaphore** существует ряд функций:

- функция **CreateSemaphore()** создает семафор с заданным начальным значением счетчика и максимальным значением, которое ограничивает доступ;
- функция **OpenSemaphore()** осуществляет доступ к семафору;
- функция **ReleaseSemaphore()** увеличивает значение счетчика. Счетчик может меняться от 0 до максимального значения;
- после завершения работы надо вызвать **CloseHandle()**.

3.3. Работа с объектом Мьютекс (Mutex)

Для работы с этим объектом есть ряд функций:

- функция создания объекта **Mutex** – **CreateMutex()**;
- для доступа – **OpenMutex()**;
- для освобождения ресурса – **ReleaseMutex()**;
- для доступа к объекту **Mutex** используется ожидающая функция **WaitForSingleObject()**.

Каждая программа создает объект **Mutex** по имени, то есть **Mutex** – это именованный объект.

П р и м е ч а н и е. Если такой объект синхронизации уже создала другая программа, то по вызову **CreateMutex()** можно получить указатель на объект, который уже создала первая программа, то есть у обеих программ будет один и тот же объект, что и позволяет производить синхронизацию.

Рассмотрим ситуацию, когда несколько процессов должны одновременно работать с одним и тем же связным списком.

Для синхронизации потоков разных процессов следует объявить один общедоступный объект класса **CMutex**, который будет управлять доступом к списку.

Однако можно подождать освобождения объекта с помощью функции **WaitForSingleObject**, в которой роль управляющего объекта выполняет тот же **Mutex**. Алгоритм использования следующий:

1) объект

CMutex mutex;

необходимо объявить заранее;

2) в точке, где необходимо защитить код, создается объект класса **CSingleLock**, которому передается ссылка на **Mutex**;

3) при попытке включения блокировки вызовом метода **Lock** надо в качестве параметра указать время (в миллисекундах), в течение которого следует ждать освобождения объекта, охраняемого **Mutex**. В течение этого времени либо получим доступ к объекту, либо нет;

4) если объект стал доступен, то запираем его от других потоков и производим работу, которая требует синхронизации. После этого снимаем блокировку;

5) если время ожидания истекло, и доступ к объекту не получен, то производится обработка этой ситуации (ветвь **else**);

6) если задать ноль в качестве параметра функции **Lock**, то ожидания не будет;

7) если передать константу **INFINITE**, можно ждать неопределенно долго;

8) другой процесс, если он знает, что существует **Mutex** с каким-то именем, может сделать этот объект доступным для себя, открыв уже существующий **Mutex**;

9) при вызове функции **OpenMutex** система сканирует существующие объекты **Mutex**, проверяя, нет ли среди них объекта с указанным именем. Обнаружив таковой, она создает описатель объекта, специфичный для данного процесса. Теперь любой поток данного процесса может использовать описатель в целях синхронизации доступа к какому-то коду или объекту;

10) когда **Mutex** становится ненужным, следует освободить его вызовом.

CloseHandle(HANDLE hObject);

где **hObject** – описатель **Mutex**.

Примечания:

1. Когда система создает **Mutex**, она присваивает ему имя. Это имя используется при совместном доступе к **Mutex** нескольких процессов.

2. Если несколько потоков создают объект с одним и тем же именем, то только первый вызов приводит к созданию **Mutex**.

3. Имя используется при совместном доступе нескольких процессов. Если оно совпадает с именем уже существующего объекта, конструктор создает новый экземпляр класса **CMutex**, который ссылается на существующий **Mutex** с данным именем.

4. Если имя не задано, **Mutex** будет неименованным, и им можно пользоваться только в пределах одного процесса.

С любым объектом ядра сопоставляется счетчик, фиксирующий, сколько раз данный объект передавался во владение потокам.

Объект **Mutex** отличается от других синхронизирующих объектов ядра тем, что занявшему его потоку передаются права на владение им.

Прочие синхронизирующие объекты могут быть либо свободны, либо заняты и только, а **Mutex** способны еще и **запоминать**, какому потоку они принадлежат.

Отказ от **Mutex** происходит, когда ожидавший его поток захватывает этот объект, переводя его в занятое состояние, а потом завершается. В таком случае получается, что **Mutex** занят и никогда не освободится, поскольку другой поток не сможет этого сделать. Система не допускает подобных ситуаций и, заметив, что произошло, автоматически переводит **Mutex** в свободное состояние.

3.4. Работа с объектом Событие (Event)

Для работы с объектом **Event** есть ряд функций:

- функция **CreateEvent()** используется для создания события;
- функция **OpenEvent()** – для доступа;
- две функции **SetEvent()** и **PulseEvent()** – для установки события;
- функция **ResetEvent()** используется для сброса события.

Дескриптор события после окончания работы нужно закрыть.

Класс **CEvent** представляет функциональность синхронизирующего объект ядра (события). Он позволяет одному потоку уведомить (*notify*) другой поток о том, что произошло событие, которое тот поток, возможно, ждал.

Существуют два типа объектов: ручной (*manual*) и автоматический (*automatic*):

- ручной объект начинает сигнализировать, когда будет вызван метод **SetEvent**. Вызов **ResetEvent** переводит его в противоположное состояние;
- автоматический объект класса **CEvent** не нуждается в сбросе. Он сам переходит в состояние *nonsignaled*, и охраняемый код при этом недоступен, когда хотя бы один поток был уведомлен о наступлении события.

4. Функции ряда Wait

Win32 API поддерживает целый ряд функций, которые начинаются с **Wait**:

- **WaitForMultipleObjects**;
- **WaitForMultipleObjectsEx**;
- **WaitForSingleObject**;
- **WaitForSingleObjectEx**;
- **MsgWaitForMultipleObjects**;
- **WaitForMultipleObjectsEx**.

Также существует функция **WaitCommEvent()**, предназначенная для работы с данными в последовательных портах.

Функции, у которых в имени есть **Single**, предназначены для установки одного синхронизирующего объекта.

Функции, у которых в имени есть **Multiple**, используются для установки ожидания сразу нескольким объектам.

Функции с префиксами **Msg** предназначены для ожидания события определенного типа, например, ввода с клавиатуры.

Функции с окончанием **Ex** расширены опциями по управлению асинхронным вводом-выводом.

П р и м е ч а н и е. При необходимости захвата нескольких ресурсов проблему решает использование **WaitForMultipleObject()**, так как эта функция, ожидая несколько объектов, пока не захватит их все, менять состояние одного из них не будет. **WaitForSingleObject()** в этом случае использовать нельзя, так как это приведет к deadlock.

4.1. Использование функций ряда Wait для синхронизации потоков

Потоки «усыпляют» себя до освобождения какого-либо синхронизирующего объекта с помощью следующих функций ряда **Wait**:

- **DWORD WaitForSingleObject (HANDLE hObject, DWORD dwTimeOut);**
- **DWORD WaitForMultipleObjects (DWORD nCount, CONST HANDLE* lpHandles, BOOL bWaitAll, DWORD dwTimeOut);**

Первая функция приостанавливает поток до тех пор, пока:

- заданный параметром **hObject** синхронизирующий объект не освободится;

- не истечет интервал времени, задаваемый параметром **dwTimeOut**. Если указанный объект в течение заданного интервала не перейдет в свободное состояние, то система вновь активизирует поток, и он продолжит свое выполнение.

В качестве параметра **dwTimeOut** могут выступать два особых значения:

0 – функция только проверяет состояние объекта (занят или свободен) и сразу же возвращается;

INFINITE – время ожидания бесконечно; если объект так и не освободится, поток останется в неактивном состоянии и никогда не получит процессорного времени.

Функция **WaitForSingleObject** в соответствии с причинами, по которым поток продолжает выполнение, может возвращать одно из следующих значений:

WAIT_TIMEOUT – объект не перешел в свободное состояние, но интервал времени истек;

WAIT_ABANDONED – ожидаемый объект является **Mutex**, который не был освобожден владеющим им потоком перед окончанием этого потока. Объект **Mutex** автоматически переводится системой в состояние свободен. Такая ситуация называется «отказ от **Mutex**»;

WAIT_OBJECT_0 – объект перешел в свободное состояние;

WAIT_FAILED – произошла ошибка, причину которой можно узнать, вызвав **GetLastError**.

Функция **WaitForMultipleObjects** задерживает поток и, в зависимости от значения флага **bWaitAll**, ждет одного из следующих событий:

- освобождение хотя бы одного синхронизирующего объекта из заданного списка;

- освобождение всех указанных объектов;

- истечение заданного интервала времени.

5. Приоритеты в Windows

5.1. Приоритеты процессов

Часть ОС, называемая системным планировщиком (**system scheduler**), управляет переключением заданий, определяя, какому из конкурирующих потоков следует выделить следующий квант времени процессора.

Решение принимается с учетом приоритетов конкурирующих потоков.

Множество приоритетов, определенных в операционной системе для потоков, занимает диапазон от 0 (низший приоритет) до 31 (высший приоритет).

Нулевой уровень приоритета система присваивает особому потоку обнуления свободных страниц. Он работает при отсутствии других потоков, требующих внимания со стороны операционной системы. Ни один поток, кроме него, не может иметь нулевой уровень.

Приоритет каждого потока определяется в два этапа, исходя из:

1) класса приоритета процесса, в контексте которого выполняется поток;

2) уровня приоритета потока внутри класса приоритета потока.

Комбинация этих параметров определяет *базовый приоритет* (*base priority*) *потока*. Существуют *шесть классов* приоритетов для процессов:

- **IDLE_PRIORITY_CLASS**;
- **BELOW_NORMAL_PRIORITY_CLASS**;
- **NORMAL__PRIORITY_CLASS**;
- **ABOVE_NORMAL_PRIORITY_CLASS**;
- **HIGH_PRIORITY_CLASS**;
- **REALTIME_PRIORITY_CLASS**.

Работа с приоритетами процесса:

- по умолчанию процесс получает класс приоритета **NORMAL_PRIORITY_CLASS**;

- программист может задать класс приоритета создаваемому им процессу, указав его в качестве одного из параметров функции **CreateProcess**;

- кроме того, существует возможность динамически, во время выполнения потока, изменять класс приоритета процесса с помощью API-функции **SetPriorityClass**;

- выяснить класс приоритета какого-либо процесса можно с помощью API-функции **GetPriorityClass**.

Процессы, осуществляющие мониторинг системы, а также хранители экрана (*screen savers*) должны иметь низший класс (**IDLE...**), чтобы не мешать другим полезным потокам.

Процессы самого высокого класса (**REALTIME...**) способны прервать даже те системные потоки, которые обрабатывают сообщения мыши, ввод с клавиатуры и фоновую работу с диском. Этот класс должны иметь только те процессы, которые выполняют короткие ***обменные операции с аппаратурой***.

Для написания драйвера какого-либо устройства, используя API-функции из набора *Device Driver Kit* (DDK), следует использовать для процесса класс **REALTIME...**

С осторожностью следует использовать класс **HIGH_PRIORITY_CLASS**, так как если поток процесса этого класса подолгу занимает процессор, то другие потоки не имеют шанса получить свой квант времени. Если несколько потоков имеют высокий приоритет, то эффективность работы каждого из них, а также всей системы резко падает. Этот класс зарезервирован для реакций на события, критичные ко времени их обработки.

Пример 1. С помощью функции **SetPriorityClass** процессу временно присваивают значение **HIGH...**, затем, после завершения **CriticalSection** кода, его снижают.

Пример 2. Создается процесс с высоким классом приоритета и тотчас же блокируется – погружается в сон с помощью функции **Sleep**. При возникновении критической ситуации поток или потоки этого процесса пробуждаются только на то время, которое необходимо для обработки события.

5.2. Приоритеты потоков

Теперь рассмотрим уровни приоритета, которые могут быть присвоены потокам процесса. Внутри каждого процесса, которому присвоен какой-либо класс приоритета, могут существовать потоки, где *уровень приоритета* принимает одно из семи возможных значений:

- **THREAD_PRIORITY_IDLE**;
- **THREAD_PRIORITY_LOWEST**;
- **THREAD_PRIORITY__BELOW___NORMAL**;
- **THREAD_PRIORITY_NORMAL**;
- **THREAD_PRIORITY_ABOVE_NORMAL**;
- **THREAD_PRIORITY_HIGHEST**;
- **THREAD_PRIORITY_TIME_CRITICAL**.

Работа с приоритетами потока следующая:

- все потоки сначала создаются с уровнем **THREAD_PRIORITY_NORMAL**;

- программист может изменить этот начальный уровень, вызвав функцию **SetThreadPriority**;

- для определения текущего уровня приоритета потока существует функция **GetThreadPriority**, которая возвращает один из семи рассмотренных уровней.

Типичной стратегией является повышение уровня до **...ABOVE_NORMAL** или **...HIGHEST** для потоков, которые должны быстро реагировать на действия пользователя по вводу информации.

Потоки, которые интенсивно используют процессор для вычислений, часто относят к фоновым. Им дают уровень приоритета **...BELOW_NORMAL** или **...LOWEST**, чтобы при необходимости они могли быть вытеснены.

Иногда возникает ситуация, когда поток с более высоким приоритетом должен ждать поток с низким приоритетом, пока тот не закончит какую-либо операцию. В этом случае не следует программировать ожидание завершения операции в виде цикла, так как львиная доля времени процессора уйдет на выполнение команд этого цикла. Возможно даже зацикливание – ситуация

типа deadlock, так как поток с более низким приоритетом не имеет шанса получить управление и завершить операцию. Обычной практикой в таких случаях является использование:

- одной из функций ожидания (*wait functions*);
- вызов функции **Sleep** (*sleepEx*);
- вызов функции **SwitchToThread**;
- использование объекта типа **Critical section** (Критическая секция).

Базовый приоритет потока является комбинацией класса приоритета процесса и уровня приоритета потока.

Ознакомьтесь с таблицей приоритетов в справке (Help), в разделе Platform SDK-Scheduling Priorities (Платформа, SDK-Планирование приоритетов).

Пример 3. Считая, что класс приоритета процесса не изменяется и остается равным **HIGH_PRIORITY_CLASS**, сведем все семь возможных вариантов в табл. 1.

Таблица 1

Приоритеты потоков

Уровень приоритета потока	Базовый уровень
THREAD_PRIORITY_IDLE	1
THREAD_PRIORITY_LOWEST	11
THREAD_PRIORITY_BELOW_NORMAL	12
THREAD_PRIORITY_NORMAL	13
THREAD_PRIORITY_ABOVE_NORMAL	14
THREAD_PRIORITY_HIGHEST	15
THREAD_PRIORITY_TIME_CRITICAL	15

5.3. Переключение потоков

Планировщик операционной системы поддерживает для каждого из *базовых уровней приоритета* функционирование *очереди* выполняемых или готовых к выполнению потоков (*ready threads queue*). Когда процессор становится доступным, то планировщик производит переключение контекстов.

Алгоритм переключения следующий:

- сохранение контекста потока, завершающего выполнение;
- перемещение этого потока в конец своей очереди;
- поиск очереди с высшим приоритетом, которая содержит потоки, готовые к выполнению;
- выбор первого потока из этой очереди, загрузка его контекста и запуск на выполнение.

Примечание. Если в системе за каждым процессором закреплен хотя бы один поток с приоритетом 31, то остальные потоки с более низким приоритетом не смогут получить доступ к процессору и поэтому не будут выполняться. Такая ситуация называется starvation.

Различают потоки, *неготовые* к выполнению:

- потоки, которые при создании имели флаг **CREATE_SUSPENDED**;
- потоки, выполнение которых было прервано вызовом функции **SuspendThread** или **SwitchToThread**;
- потоки, которые ожидают ввода или синхронизирующего события.

Блокированные таким образом потоки, или *приостановленные* (*suspended*) потоки, не получают кванта времени независимо от величины их приоритета.

Типичные причины переключения контекстов:

- истек квант времени;
- в очереди с более высоким приоритетом появился поток, готовый к выполнению;
- текущий поток вынужден ждать.

В последнем случае система не ждет завершения кванта времени и отбирает управление, как только поток впадает в ожидание. Каждый поток обладает *динамическим* приоритетом, кроме рассмотренного базового уровня. Под этим понятием скрываются *временные колебания* уровня приоритета, которые вызваны планировщиком. Он намеренно вызывает такие колебания для того, чтобы убедиться в управляемости и реактивности потока, а также, чтобы дать шанс выполниться потокам с низким приоритетом (система никогда не подстегивает потоки, приоритет которых и так высок (от 16 до 31)).

Когда пользователь работает с каким-то процессом, то он считается активным (*foreground*), а остальные процессы – фоновыми (*background*). При ускорении потока (*priority boost*) система действует следующим образом: когда процесс с нормальным классом приоритета «выходит на передний план» (*is brought to the foreground*), он получает ускорение.

Примечание. Термин «foreground» обозначает качество процесса, которое характеризует его с точки зрения связи с активным окном Windows. Foreground window – это окно, которое в данный момент находится в фокусе и, следовательно, расположено поверх остальных. Это состояние может быть получено как программным способом (вызов функции **SetFocus**), так и аппаратно (пользователя в окне).

Планировщик изменяет класс процесса, связанного с этим окном, так, чтобы он был больше или равен классу любого процесса, связанного с *background*-окном. Класс приоритета вновь восстанавливается при потере процессом статуса *foreground*. Пользователь может управлять величиной ускорения всех процессов класса **NORMAL_PRIORITY** с помощью панели управления (команда System, вкладка Performance, ползунок Boost Application Performance).

Когда окно получает сообщение типа **WM_TIMER**, **WM_LBUTTONDOWN** или **WM_KEYDOWN**, планировщик также ускоряет (*boosts*) поток, владеющий этим окном.

Существуют еще ситуации, когда планировщик временно повышает уровень приоритета потока. Довольно часто потоки ожидают возможности обратиться к диску. Когда диск освобождается, блокированный поток просыпается, и в этот момент система повышает его уровень приоритета. После ускорения потока планировщик постепенно снижает уровень приоритета до базового значения. Уровень снижается на одну единицу после завершения очередного кванта времени.

Иногда система инвертирует приоритеты, чтобы разрешить конфликты типа deadlock. Благодаря динамике изменения приоритетов потоки активного процесса вытесняют потоки фонового процесса, а потоки с низким приоритетом все-таки имеют шанс получить управление.

Пример 4.

Поток1 с высоким приоритетом вынужден ждать, пока Поток2 с низким приоритетом выполняет код в критической секции. В это же время готов к выполнению Поток3 со средним значением приоритета. Он получает время процессора, а два других потока застревают на неопределенное время, так как Поток2 не в состоянии вытеснить Поток3, а Поток1 помнит, что надо ждать, когда Поток2 выйдет из критической секции.

Операционная система Windows разрешает эту ситуацию так: планировщик увеличивает приоритеты *готовых* потоков на величину, выбранную случайным образом. В нашем примере это приводит к тому, что поток с низким приоритетом получает шанс на время процессора и в течение, может быть, нескольких квантов закончит выполнение кодов критической секции. Как только это произойдет, Поток1 с высоким приоритетом сразу получит управление и сможет, вытеснив Поток3, начать выполнение кодов критической секции.

Программист имеет возможность управлять процессом ускорения потоков с помощью API-функций **SetProcessPriorityBoost** (все потоки данного процесса) или **SetThreadPriorityBoost** (данный поток). Функции **GetProcessPriorityBoost** и **GetThreadPriorityBoost** позволяют выяснить текущее состояние флага.

При наличии нескольких процессоров Windows применяет симметричную модель распределения потоков по процессорам *symmetric multiprocessing* (SMP). Это означает, что любой поток может быть направлен на любой процессор.

Программист может ввести некоторые коррективы в эту модель равноправного распределения. Функции **SetProcessAffinityMask** и **SetThreadAffinityMask** позволяют указать предпочтения в смысле выбора процессора для всех потоков процесса или для одного определенного потока. Потоковое предпочтение (thread affinity) вынуждает систему выбирать процессоры только из множества, указанного в маске.

Существует также возможность для каждого потока указать один предпочтительный процессор. Это делается с помощью функции **SetThreadidealProcessor**. Указание служит подсказкой для планировщика заданий.