中国科学院软件研究所

一九九六年招收硕士学位研究生入学考试试题

试题名称：软件基础

一.（10 分）

1. 给出下列表达式的逆波兰表示（后缀式）:（各 1 分）

   a+b*c;

   a≦b+c∧a>d∨a+b≠e

2. 写出下列语句的逆波兰式表示（后缀式）:（各 1 分）

   < 变量 > : = < 表达式 >

   IF < 表达式 > THEN < 语句 1 > ELSE < 语句 2 >

3. 写出算术表达式

   A+B*（C-D）+E/（C-D）**N

   的四元式、三元式和间接三元式序列。（各 2 分）

二.（10 分）写一文法，使其语言是偶数的集合，但不允许有以 0 居首的偶整数。

三.（10 分）LRU 算法的基本思想是什么？有什么缺点？给出该算法的流程图。

四.（10 分）设有一个具有 n 个信息元素的环形缓冲区，A 进程顺序地把信息写入缓冲区，B 进程依次地从缓冲区读出信息，回答下列问题：

1. 叙述 A、B 两进程的相互制约关系；
2. 判别下列用 P、V 操作表示的同步算法是否正确？如果不正确，试说明理由，并修改成正确算法。

```
var buffer: array 0..N-1 of T;
    in,out: 0..N-1;
var S1,S2:Semaphore;
S1:==0;     s2:=N;
in:=out:=0;

Procedure A:
  begin
    repeat
      生产数据 m;
      P(S2);
      buffer(in):=m;
      in:=(in+1) mod N;
      V(S1);
    forever
  end
```

```
Procedure B:
  begin
    repeat
      V(S2);
      m:=buffer(out);
      消费 m;
      out:=(out+1) mod N;
      P(S1);
    forever
  end
```

五.（10 分）试简述 UNIX 的文件读写过程。

六.（10 分）试给出运算变量均为整数的简单算术表达式所需最少临时单元个数的算法（假定不许用代数规则变更表达式的计值顺序）。

例如：A+B*C 需要一个临时单元为（B*C）;（A+B）*（C+D）+E 则需要两个临时单元，一个为 C+D，另一个既为 A+B 又为（A+B）*（C+D）。

七.（10 分）已知三维空间（直角坐标系）有 n 个点，请编写一个函数过程，求通过某一特定平面的点数。

八.（13 分）编写一过程，对一个 n×n 矩阵，通过行变换，使其每行元素的平均值按递增顺序排列。

九.（17 分）请编写一过程，对具有 n 个整数的序列进行二叉树排序。

中国科学院软件研究所

一九九六年招收硕士学位研究生入学考试试题答案

试题名称：软件基础

一．1.　　表达式　　后缀式

①．a+b*c　　abc*+

②．a≦b+c∧a>d∨a+b≠e　　abc+≦ad>∧ab+e≠∨

2．①．赋值语句＜变量＞：=＜表达式＞的逆波兰式表示为：

＜变量＞＜表达式＞：=

②．条件语句 IF＜表达式＞THEN＜语句 1＞ELSE＜语句 2＞的逆波兰式表示为：

＜表达式＞＜语句 1＞＜语句 2＞IF THEN ELSE

或＜表达式＞P_1 Jez＜语句 1＞P_2 J P_1：＜语句 2＞P_2；，其中 Jez 是＜表达式＞和 P_1 这两个运算对象的二元运算符，表示当＜表达式＞等于 0 即取假值时转去执行由 P_1 开始的＜语句 2＞。否则，执行＜语句 1＞然后转至 P_2 所指地方；J 是无条件转移的一元运算符。

3．　①．四元式序列：

(1)　(－　C　D　T_1)
(2)　(＊　B　T_1　T_2)
(3)　(＋　A　T_2　T_3)
(4)　(－　C　D　T_4)
(5)　(＊＊ T_4　N　T_5)
(6)　(／　E　T_5　T_6)
(7)　(＋　T_3　T_6　T_7)

②．三元式序列：

(1)　(－　C　D)
(2)　(＊　B　(1))
(3)　(＋　A　(2))
(4)　(－　C　D)
(5)　(＊＊ (4)　N)
(6)　(／　E　(5))
(7)　(＋　(3)　(6))

③．间接三元式序列：

(1)　(－　C　D)
(2)　(＊　B　(1))
(3)　(＋　A　(2))
(4)　(＊＊ (1)　N)
(5)　(／　E　(4))
(6)　(＋　(3)　(5))

间接码表

(1)
(2)
(3)
(4)
(5)
(6)

二．解答：满足题意要求的文法 G 为：

G ={V_T，V_N，＜头为非零元整偶数＞，ρ}

其中，ρ：＜头为非零元整偶数＞→ ＜偶数字 1＞｜

＜非零数字＞＜偶数字＞｜

＜非零数字＞＜数＞＜偶数字＞

＜偶数字＞　→ 0｜2｜4｜6｜8

＜非零数字＞ → 1｜2｜3｜4｜5｜6｜7｜8｜9

< 数 > → < 数字 > | < 数 > < 数字 >
< 数字 > → < 非零数字 > | 0
< 偶数字 1 > → 2 | 4 | 6 | 8

V_N={ < 头为非零元整偶数 > 、 < 偶数字 > 、 < 非零数字 > 、 < 数 > 、 < 数字 > }

V_T={0， 1， 2， 3， 4， 5， 6， 7， 8， 9}

或者

G = (V_T, V_N, N, ρ),

其中ρ：N → AD’|D’ （或N→D’|DD’|DAD’）
A → AD”|D （或A→AD”|D”）
D’ → 2|4|6|8
D → 1|3|5|7|9|D’
D” → **0**|D;

V_N={N,A,D’,D,D”}

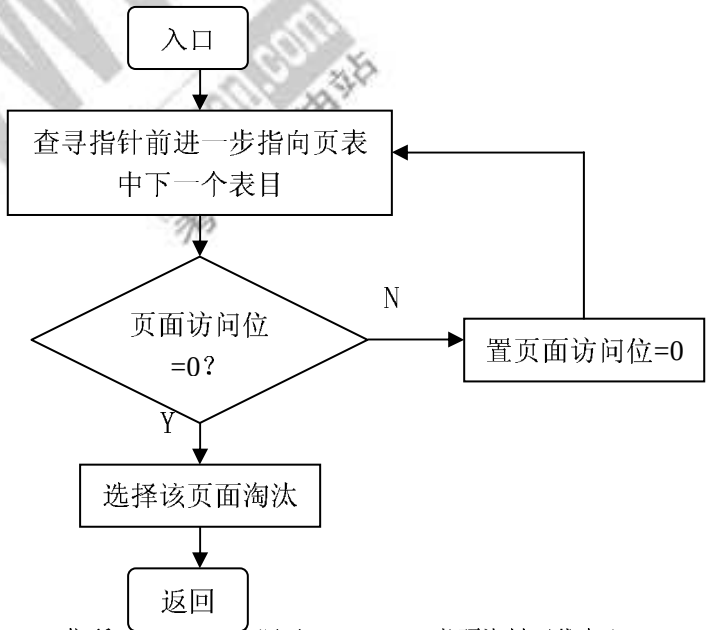
V_T={0,1,2,3,4,5,6,7,8,9},

N—开始符号。

三．LRU 算法的基本思想是：利用局部性原理，根据一个作业在执行过程中过去的页面访问踪迹来推测未来的行为。它认为过去一段时间不曾被访问过的页面，在最近的将来可能也不会再被访问。所以该算法的实质是：当需要置换一页面时，选择在最近一段时间内最久不用的页面予以淘汰。

缺点：实现起来比较困难，因为要对先前的访问历史时时加以记录和更新。如果这种连续的修改完全由软件来做，系统开销太大；如编硬件执行，则会增加机器成本。因此，在实际应用中得到推广的是一些简单而有效的 LRU 算法。

近似 LRU 算法流程图：



四. 1. 当缓冲区满时，A 进程不可以写，必须等待；
当缓冲区空时，B 进程不可以读，必须等待。

2. 该算法有错，它对读进程进入临界区未加限制，当缓冲区为空时，也可以进入临界区读信息；
当存在多个读进程多个写进程时，还需引入一个信号量 S_0 以防止同时读或同时写。

```
var  S0, S1, S2: semaphore: =1, n, 0;
       buffer: array [0..n-1] of message;
       in,out: 0..n-1:=0,...0;
begin
  parbegin
    producer: begin
                repeat
                  ┆
                  produce a new message m;
                  ┆
                  P(S1);
                  P(S0);
                  buffer(in):=m;
                  in:=(in+1) mod n;
                  V(S0);
                  V(S2);
                until false
              end
    consumer: begin
                repeat
                  P(S2);
                  P(S0);
                  m:=buffer(out);
                  out:=(out+1) mod n;
                  V(S0);
                  V(S1);
                  consume message m;
                until false
              end
  parend
end
```

五. 解答：当用户创建或联接了一个文件并把它打开后,便可以对它执行读、写操作。文件系统在进行读写操作时，需调用一系列读写有关的过程，如 (1) passc 过程、cpass 过程。前者用于把一字符从缓冲区送到用户区，后者相反； (2) iomove 过程用于实现用户区和缓冲区之间的信息传送； (3) readi 过程用于把信息从磁盘读入内存； (4) writei 过程用于把信息从

内存写入磁盘。

或：

1. 读方式

在UNIX系统中有两种读方式：⑴ 一般读方式。把盘块中的信息读入缓冲区，有bread过程完成；⑵ 提前读方式。在一个进程顺序地读入一个文件的各个盘块时，会预见到所要读的下一个盘块，因而在请求读出指定盘块（作为当前块）的同时，可要求提前将下一个盘块（提前块）中的信息读入缓冲区。这样，当以后需要该盘块的数据时，因它已在内存中，这就缩短了读数据时间，从而改善了系统性能。提前读功能由breada过程完成。

2. 写方式

UNIX系统有三种方式：⑴ 一般写方式。真正把缓冲区中的数据写入磁盘上，且进程须等待写操作完成，由过程bwrite完成；⑵ 异步写方式。进程无须等待写操作完成便可返回，异步写过程为bawrite；⑶ 延迟写方式。该方式并不真正启动磁盘，而只是在缓冲首部设置延迟写标志，然后便释放该缓冲区，并将该缓冲区链入空闲链表的末尾，以后当有进程申请到该缓冲区时，才将它写入磁盘。引入延迟写的目的是为了减少不必要的磁盘I/O，因为只要没有进程申请到此缓冲区，其中的数据便不会写入磁盘，倘若再有进程需要访问其中的数据时，便可直接从空闲链表中摘下该缓冲区，而不必从磁盘读入。

六．解答：

设运算变量均为整数的某个简单算术表达式，已加工的为k个四元式，n1是起始四元式号码，nk是终止四元式号码。由于是加工简单算术表达式得到的四元式序列，因此这些四元式均为一目或二目算术运算的四元式。为使存放中间结果的临时单元个数最少，我们设立一个计数器count。当四元式的第2或第3项出现临时变量（即出现对临时变量的引用时），每出现一个，计数器就减少1；当四元式的第4项（即存放运算结果的项）出现临时变量时，计数器就应增加1。

正如前面所述，这些四元式是加工简单算术表达式得到的算术运算四元式，因此每个四元式必然要把计算结果赋予某个临时变量，因此四元式的第4项一定是临时变量。

于是对每个算术运算四元式来说，若四元式第2，第3项都是临时变量时，计数器要减少1；当第2，第3项只有一个是临时变量，计数器不变；当第2，第3项都不是临时变量时，计数器增加1。

下面给出的算法顺便也给出了临时变量分配的临时单元地址（假定当前分配的临时变量的起始地址为a）。

```
Procedure PT (a, n1, nk);
  begin
    count: =0;    p: =0;    /* p 是临时变量个数*/
```

```
        for i：=n1 to nk do
          begin
            if 四元式的第 2、3 项都是临时变量
               then  count：=count-1;
            if 四元式的第 2、3 项都不是临时变量
               then  count：=count+1;
            p：=max (p，count);
            T[i].ADDR：=a + count －1    /* 保证起始地址为 a */
          end
      end
```

其中 p 记录这 k 个四元式所需临时单元的最大个数；count 是计数器。

例如：设有整型的算术表达式为

A*B-C*D+E*F

生成的四元式以及所需最少临时单元如下表所示

四元式	地址	count	p
(*, A, B, T_1)	a	1	1
(*, C, D, T_2)	a+1	2	2
(-, T_1,T_2, T_3)	a	1	2
(*, E, F, T_4)	a+1	2	2
(+, T_3,T_4, T_5)	a	1	2

七.

```
#define EPSINO 1.0E-10

int pass_plane(points,pno,plane)
    float * points;              /* points[pno*3] */
    int     pno;
    float   plane[4];            /* a,b,c,d for plane agu.*/
 {
   int i;
   int pass_no=0;
   float x,y,z;
   float * cfp;

   for(i=0;i<pno;i++)
     {
       cfp = points + (i*3)
       x = *cfp;    y = *(cfp+1);    z = *(cfp+2);
       if ( (plane[0]*x+plane[1]*y+plane[2]*z+plane[3]) < EPSINO)
         pass_no++;
      }
   return(pass_no);
 }
```

八.

```
Trans (mat, no)
 float *mat;                          /* matrix[no][no] */
 int    no;
 {
   int i, j, k, l;
   float sum, minf;
   float *p, *pt, *pk, *pi;

   p = (float *)  malloc (no*sizeof (float));

   for (i=0; i<no;i++)                /* sum up each row */
     {
       sum = 0.0;
       pt = mat + i*no;
       for(j=0;j<no;j++)
         {
          pt++;
          sum += (*pt);
         }
       *(p+i) = sum;
     }

   for(i=0;i<no-1;i++)
     {
       minf=*(p+i);      k=i
       for(j=i+1;j<no;j++)          /* seek for the index of mininum */
         if (minf>p[j])      {k = j;    minf = p[j];}
       if (k!=i)                        /* exchange the rows */
         {
          ph= mat + (no+k);
          pi= mat + (no*i);
          for(l=0;l<no;l++)             /* row k ⇔ row i */
            {
              sum = *(pi+l);
              *(pi+l) = *(pk+l);
              *(ph+l) = sum;
            }
          sum = p[i]; p[i]=p[k]; p[k]=sum; /* p[k] ⇔ p[i] */
         }
     } /* end of for i */
   free(p);
 }
```

九.

```
struct bintree
  {
    int ele;
    struct bintree *left;
    struct bintree *right;
  }

binsort(a,n,p_tree);              /* procedure START */
int *a;                           /* a[n] */
int n;
struct bintree * p_tree;          /* p_tree[n] */
{
  int i;
  boolean greater;
  float value;
  struct bintree *p_tree1,*p_treesave;

  for(i=0;i<n;i++)
    {
      p_tree[i].ele = a[i];
      p_tree[i].left = NULL;
      p_tree[i].right = NULL;
    }

  for(i=1;i<n;i++)
    {
      value = a[i];
      p_tree1 = p_tree;           /* root of bin_tree */
      while(p_tree1 != NULL)
        {
          p_treesave = p_tree1;
          if (greater = (p_tree1->ele > value))
             p_tree1 = p_tree1->left;
          else
             p_tree1 = p_tree1->right;
        }
      if (greater)
        p_treesave->left = p_tree+i;   /* chained onto the tree */
      else
        p_treesave->right = p_tree+i;
    }
}
```