

# 实验报告五 文件管理

## 一、 实验目的

- 掌握文件系统的工作机理。
- 掌握各种文件管理算法的实现方法
- 通过实验比较各种文件管理算法的优劣。

## 二、 相关知识

- 位表、链式空闲区、索引和空闲块列表算法。
- 操作系统相关文件管理知识。
- 在本次文件管理实验中，在内存中模拟一个磁盘环境，并实现一个文件管理算法，用于管理模拟的磁盘空间。同时，在这个模拟的磁盘环境中实验文件的创建，删除等操作。

## 三、 实验内容

给出一个磁盘块序列：1、2、3、.....、500，初始状态所有块为空的，每块的大小为 2k。选择使用位表、链式空闲区、索引和空闲块列表四种算法之一来管理空闲块。对于基于块的索引分配执行以下步骤：

随机生成 2k-10k 的文件 50 个，文件名为 1.txt、2.txt、.....、50.txt，按照上述算法存储到模拟磁盘中。

删除奇数.txt（1.txt、3.txt、.....、49.txt）文件

新创建 5 个文件（A.txt、B.txt、C.txt、D.txt、E.txt），大小为：7k、5k、2k、9k、3.5k，按照与（1）相同的算法存储到模拟磁盘中。

给出文件 A.txt、B.txt、C.txt、D.txt、E.txt 的文件分配表和空闲

区块的状态。

## 四、 实验环境

- PC + Linux Red Hat 操作系统 + GCC
- 或 Windows xp + VC
- 或 任意 OS + Java

## 五、 实验中遇到的主要问题及其解决方法

无特别大问题。

## 六、 源代码和流程图

基于长度可变区域的索引分配 + 位表



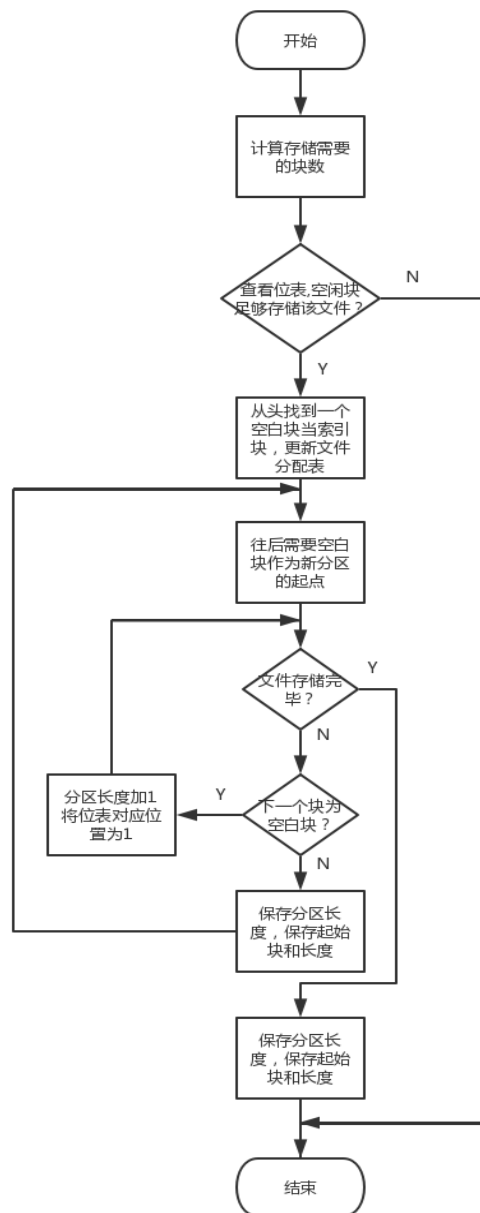
main.cpp

- ✧ 随机生成 2k-10k 的文件 50 个，文件名为 1.txt、2.txt、.....、50.txt，并将其模拟上述算法存储到模拟磁盘中。

思路：

- 随机产生文件大小
- 计算存储需要的磁盘块数，并根据位表判断是否有足够的磁盘空间
- 找出一个空闲块当做索引块，将对应位表位置为1
- 找到多个分区，记录下分区的起始块和长度，并将对应位表位置为1，将文件全部存储完毕

流程图：



代码：

```
for (int i = 0; i < 50; i++) {
    fileName = to_string(i+1) + ".txt"; //文件名
    fileSize = randFileSize(); //随机产生文件大小
    file[i].name = fileName;
    file[i].size = fileSize;
    //存储该文件
    if (storageFile(bitTable, FAT, file[i], disk)) {
        outputIndexBlock(disk, FAT, file[i]); cout << endl; //输出文件分配表
    }
}
```

```

/*
    存储文件
*/
bool storageFile(int bitTable[], vector<Fat> &FAT, File file, int disk[][LEN])
{
    int n = ceil(file.size/2); //向上取整，获得文件存储需要的块数（不包括索引块）

    //检查是否有足够的空块来存放该文件
    int i = 0;
    int k = 0;
    int index = -1; //索引块
    for (; i < LEN; i++) {
        if (bitTable[i] == 0) {
            k++;
            if (k == 1) {
                index = i; //记下索引块
            }
            if (k == n + 1) {
                break;
            }
        }
    }
    //没有足够空间来存放该文件
    if (k <= n) {
        return false;
    }
    //更新文件分配表
    Fat fat;
    fat.fileName = file.name;
    fat.indexBlock = index;
    FAT.push_back(fat);

    //更新位表
    bitTable[index] = 1;
    i = index + 1; //从索引块的下一个位置继续寻找空白块
    k = 0; //记录每段分区的长度
    int t = 0;
    //根据位表找到存放的位置，更新索引块，更新位表
    for (int j = 0; j < n; j++) {
        if (bitTable[i] == 0) {
            if (k == 0) { //这是一个新的分区，记录下起始块
                disk[index][t++] = i;
            }
            k++;
        }
    }
}

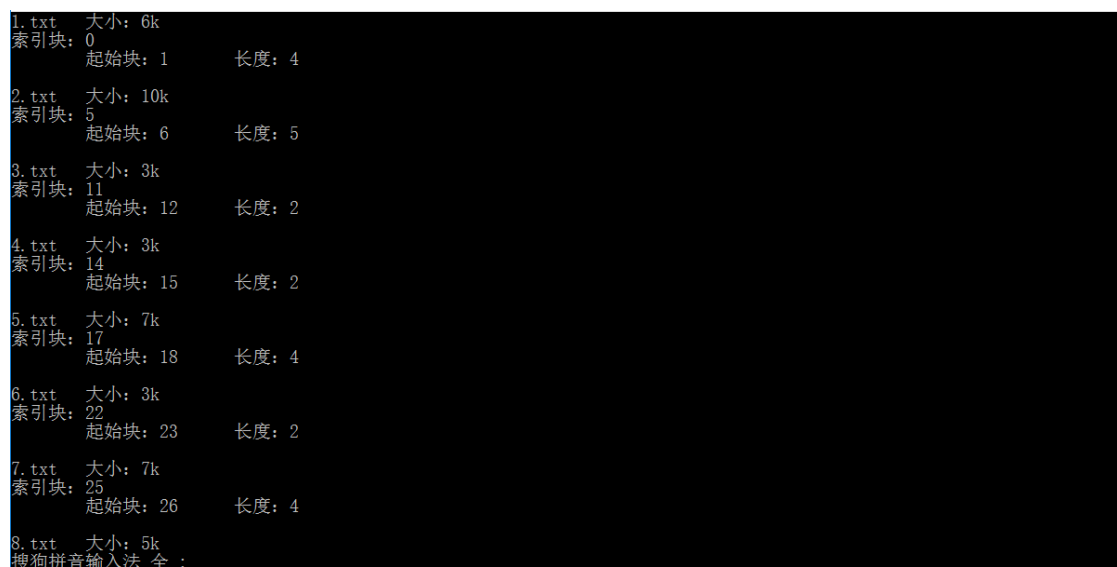
```

```

        bitTable[i] = 1; //更新位表
    }
    else{
        if (k != 0) { //如果 k!=0 说明，该分区结束，记录下长度
            disk[index][t++] = k;
            disk[index][t] = -1; //设置一个标志
            k = 0;
        }
        j--;
    }
    i++;
}
disk[index][t++] = k;
disk[index][t] = -1;
return true;
}

```

运行截图：



#### ✧ 删除文件：

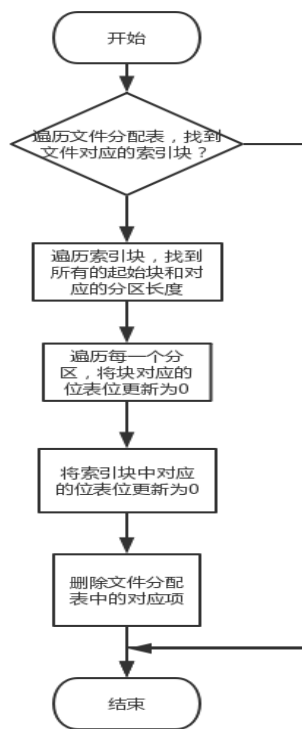
思路：a. 在文件分配表中找到对应的索引块

b. 找出索引块中的起始块和长度，将对应位表位重置为 0

c. 将索引块对应的位表位重置为 0

d. 删除文件分配表中对应项

流程图：



代码：

```

for (int i = 1; i < 50; i += 2) {
    string fileName = to_string(i) + ".txt";
    if(!deleteFile(bitTable, FAT, fileName, disk))
        cout << "成功删除文件：" << fileName << endl;
}

/*
    删除文件更新位表
*/
bool deleteFile(int bitTable[], vector<Fat> &FAT, string fileName, int disk[][LEN])
{
    //从文件分配表中找到索引块
    for (int i = 0; i < FAT.size(); i++) {
        if (FAT[i].fileName == fileName) {
            int index = FAT[i].indexBlock;
            int t = 0;
            while (disk[index][t] != -1) {
                int begin = disk[index][t];
                int len = disk[index][t + 1];
                for (int j = 0; j < len; j++) {
                    bitTable[begin + j] = 0; //重置对应位表位
                }
            }
        }
    }
}

```

```

        }
        t += 2;
    }
    bitTable[index] = 0; //重置索引块对应位表位
    vector<Fat>::iterator itr = FAT.begin()+i;
    FAT.erase(itr); //删除文件分配表对应项
    return true;
}
}
return false;
}

```

实验结果：

```

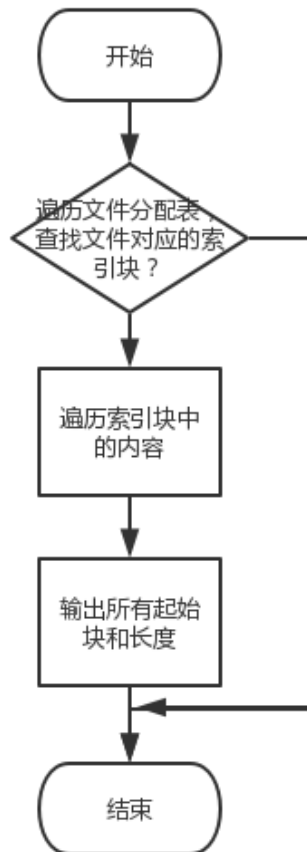
成功删除文件: 1.txt
成功删除文件: 3.txt
成功删除文件: 5.txt
成功删除文件: 7.txt
成功删除文件: 9.txt
成功删除文件: 11.txt
成功删除文件: 13.txt
成功删除文件: 15.txt
成功删除文件: 17.txt
成功删除文件: 19.txt
成功删除文件: 21.txt
成功删除文件: 23.txt
成功删除文件: 25.txt
成功删除文件: 27.txt
成功删除文件: 29.txt
成功删除文件: 31.txt
成功删除文件: 33.txt
成功删除文件: 35.txt
成功删除文件: 37.txt
成功删除文件: 39.txt
成功删除文件: 41.txt
成功删除文件: 43.txt
成功删除文件: 45.txt
成功删除文件: 47.txt
成功删除文件: 49.txt

```

✧ 输出文件 A.txt B.txt C.txt D.txt E.txt 对应的文件分配表和索引块内容：

- 思路：
- 在文件分配表中找到对应的索引块
  - 输出索引块中的所有起始块和长度

流程图：



代码：

```
/*  
    输出索引块和存储的位置  
*/  
void outputIndexBlock(int disk[][LEN], vector<Fat> FAT, File file)  
{  
    string fileName = file.name;  
    int size = file.size;  
    cout << fileName << "\t大小: " << size << "k" << endl;  
    for (int i = 0; i < FAT.size(); i++) {  
        if (FAT[i].fileName == fileName) {  
            int index = FAT[i].indexBlock;  
            cout << "索引块: " << index << endl;  
            int t = 0;  
            while (disk[index][t] != -1) {  
                int begin = disk[index][t];  
                int len = disk[index][t + 1];  
                cout << "\t起始块: " << begin << "\t长度: " << len << endl;  
            }  
        }  
    }  
}
```



```

        t += 2;
    }
    break;
}
}
}

```

执行结果：

```

-----
文件分配表中:
A.txt 大小: 7k
索引块: 0
      起始块: 1      长度: 4

B.txt 大小: 5k
索引块: 11
      起始块: 12      长度: 2
      起始块: 17      长度: 1

C.txt 大小: 2k
索引块: 18
      起始块: 19      长度: 1

D.txt 大小: 9k
索引块: 20
      起始块: 21      长度: 1
      起始块: 25      长度: 4

E.txt 大小: 3k
索引块: 29
      起始块: 34      长度: 2

```

#### ✧ 输出空闲块：

思路：a. 遍历位表

b. 找到所有为 0 的位表位，并输出

代码：

```

/*
打印空白块
*/
void outputFreeBlock(int bitTable[])
{
    cout << "空闲块" << endl;
    for (int i = 0; i < LEN; i++) {
        if (bitTable[i] == 0) {
            cout << i << "\t";
        }
    }
    cout << endl;
}

```

运行结果：

空闲块														
36	37	38	39	43	44	45	46	51	52	53	54	60	61	62
63	64	65	69	70	71	72	73	74	79	80	81	82	83	87
88	89	90	91	92	97	98	99	106	107	108	109	110	117	118
119	123	124	125	126	132	133	134	138	139	140	141	146	147	148
155	156	157	158	159	160	166	167	168	169	173	174	175	176	183
184	185	186	187	192	193	194	200	201	202	203	204	205	209	210
211	212	213	214	221	222	223	224	225	226	227	228	229	230	231
232	233	234	235	236	237	238	239	240	241	242	243	244	245	246
247	248	249	250	251	252	253	254	255	256	257	258	259	260	261
262	263	264	265	266	267	268	269	270	271	272	273	274	275	276
277	278	279	280	281	282	283	284	285	286	287	288	289	290	291
292	293	294	295	296	297	298	299	300	301	302	303	304	305	306
307	308	309	310	311	312	313	314	315	316	317	318	319	320	321
322	323	324	325	326	327	328	329	330	331	332	333	334	335	336
337	338	339	340	341	342	343	344	345	346	347	348	349	350	351
352	353	354	355	356	357	358	359	360	361	362	363	364	365	366
367	368	369	370	371	372	373	374	375	376	377	378	379	380	381
382	383	384	385	386	387	388	389	390	391	392	393	394	395	396
397	398	399	400	401	402	403	404	405	406	407	408	409	410	411
412	413	414	415	416	417	418	419	420	421	422	423	424	425	426
427	428	429	430	431	432	433	434	435	436	437	438	439	440	441
442	443	444	445	446	447	448	449	450	451	452	453	454	455	456
457	458	459	460	461	462	463	464	465	466	467	468	469	470	471
472	473	474	475	476	477	478	479	480	481	482	483	484	485	486
487	488	489	490	491	492	493	494	495	496	497	498	499		

## 七、实验总结

- ✧ 通过实验实现基于长度可变区域的索引分配 + 位表的文件管理，我对文件管理有了更加深刻理解。