廣東工業大學

实 验 报 告

课程名称____操作系统实验_____

学生学院____先进制造学院_____

专业班级_____计算机科学与技术 6 班_

学　　号_____3123008647_____

学生姓名_____林峻安_____

指导教师____苏畅、李剑锋_____

2025 年　　6 月　　1　　日

实验一 进程/作业调度

一、实验目的

二、实验内容

三、实现思路

四、主要的数据结构

五、算法流程图

六、运行与测试

七、改进的方向

实验二 动态分区分配方式的模拟

一、实验目的

了解动态分区分配方式中使用的数据结构和分配算法，并进一步加深对动态分区存储管理方式及其实现过程的理解。

二、实验内容

1.算法实现:

A.使用 C 语言实现动态分区分配的三种核心算法：

首次适应算法 (First_fit())

最佳适应算法 (Best_fit())

最坏适应算法 (Worst_fit())

B.实现通用的内存回收函数 (Free_mem())。

C.要求：使用**空闲分区链表**来管理空闲内存区域。此链表需按照分区的**起始地址从低到高**进行排序。

2.模拟与验证:

设定初始状态：总可用内存空间为 640KB。

A. 模拟处理以下作业请求与释放序列：

作业 1 申请 130KB

作业 2 申请 60KB

作业 3 申请 90KB

作业 2 释放内存 (60KB)

作业 4 申请 200KB

作业 3 释放内存 (90KB)

作业 1 释放内存 (130KB)

作业 5 申请 140KB

作业 6 申请 60KB

作业 7 申请 50KB

作业 6 释放内存 (60KB)

作业 8 申请 170KB

B. 要求：分别使用上述三种算法，跟踪并记录（例如，通过绘制内存分区图或列表形式）在处理序列中**每一步**操作后的内存分配状况（包括已分配区域和空闲区域列表）。

三、实现思路

四、主要的数据结构

```
typedef int Status;
int flag;
```

这是一个简单的类型定义，将 `int` 类型重命名为 `Status`，通常用于表示函数返回状态（如成功/失败）`flag` 是一个全局整型变量，可能用于状态标记

```
typedef struct freearea
{
    long size;      // 分区大小
    long address;   // 分区地址
    int state;      // 状态
} ElemType;
```

定义了一个表示内存空闲区域的结构体

包含区域大小、起始地址和当前状态信息

```
typedef struct DuLNode
{
    ElemType data;            // 数据域
    struct DuLNode *prior;    // 前趋指针
    struct DuLNode *next;     // 后继指针
} DuLNode, *DuLinkList;
```

定义了双向链表的节点结构

每个节点包含：

1.data 字段(ElemType 类型)：存储空闲区域信息

2.prior 指针：指向前一个节点

3.next 指针：指向后一个节点

4.DuLinkList 是指向此节点的指针类型

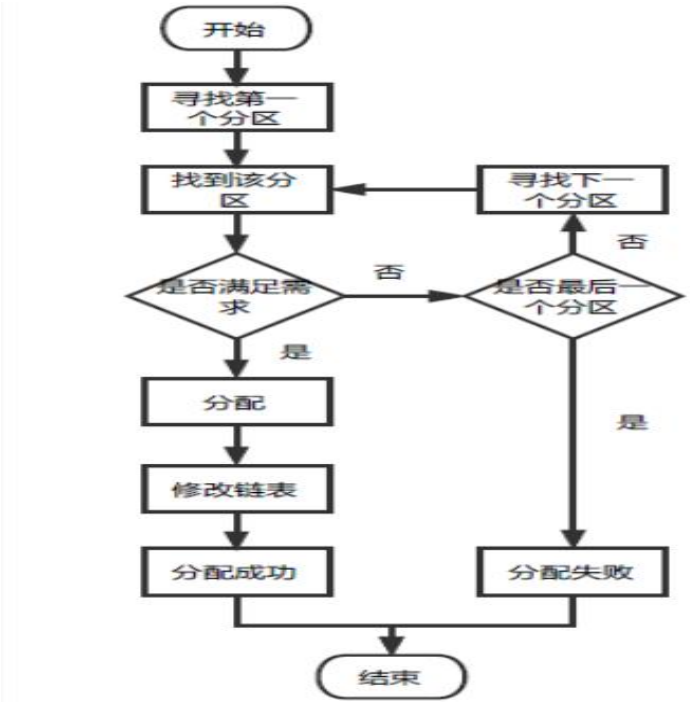
- block_first 指向双向链表的第一个节点
- block_last 指向双向链表的最后一个节点

五、算法流程图

First_fit()及其流程图：

```
//首次适应算法
Status First_fit(int request)
{
    //为申请作业开辟新空间且初始化
    DuLinkList temp = (DuLinkList)malloc(sizeof(DuLNode));
    temp->data.size = request;
    temp->data.state = Busy;
    DuLNode *p = block_first->next;
    while (p)
    {
        if (p->data.state == Free && p->data.size == request)
        { //有大小恰好合适的空闲块

            //  待补全
            p->data.state=Busy;
            return OK;
            break;
        }
        if (p->data.state == Free && p->data.size > request)
        { //有空闲块能满足需求且有剩
            temp->prior=p->prior;
            temp->next=p;
            temp->data.address=p->data.address;
            p->prior->next=temp;
            p->prior=temp;
            p->data.address=temp->data.address+temp->data.size;
            p->data.size-=request;
            return OK;
            break;
        }
        p = p->next;
    }
    return ERROR;
```

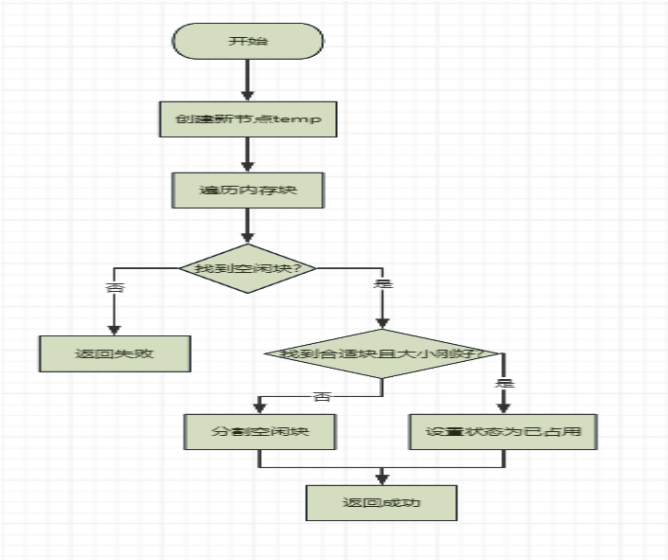


最佳适应算法（Best_fit()）函数代码及其流程图：

```
//最佳适应算法
Status Best_fit(int request)
{
    int ch; //记录最小剩余空间
    DuLinkList temp = (DuLinkList)malloc(sizeof(DuLNode));
    temp->data.size = request;
    temp->data.state = Busy;
    DuLNode* p = block_first->next;
    DuLNode* q = NULL; //记录最佳插入位置

    while (p) //初始化最小空间和最佳位置
    {
        if (p->data.state == Free && (p->data.size >= request))
        {
            if (q == NULL)
            {
                q = p;
                ch = p->data.size - request;
            }
            else if (q->data.size > p->data.size)
            {
                q = p;
                ch = p->data.size - request;
            }
        }
        p = p->next;
    }

    if (q == NULL)
        return ERROR; //没有找到空闲块
    else if (q->data.size == request)
    {
        q->data.state = Busy;
        return OK;
    }
    else
    {
        temp->prior = q->prior;
        temp->next = q;
        temp->data.address = q->data.address;
        q->prior->next = temp;
        q->prior = temp;
        q->data.address += request;
        q->data.size = ch;
        return OK;
    }
    return OK;
}
```



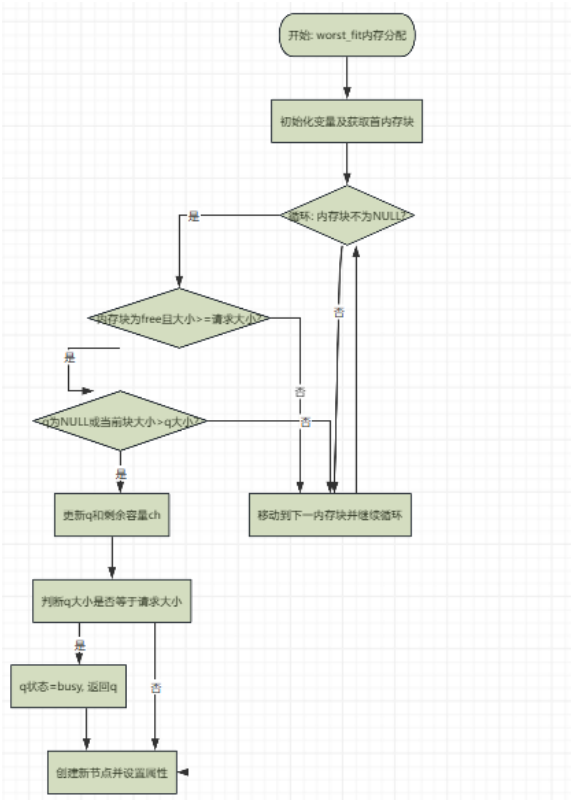
1

最坏适应算法（Worst_fit()）函数代码及其流程图：

```
//最差适应算法
Status Worst_fit(int request)
{
    int ch; //记录最大剩余空间
    DuLinkList temp = (DuLinkList)malloc(sizeof(DuLNode));
    temp->data.size = request;
    temp->data.state = Busy;
    DuLNode* p = block_first->next;
    DuLNode* q = NULL; //记录最佳插入位置

    while (p) //初始化最大空间和最佳位置
    {
        if (p->data.state == Free && (p->data.size >= request))
        {
            if (q == NULL)
            {
                q = p;
                ch = p->data.size - request;
            }
            else if (q->data.size < p->data.size)
            {
                q = p;
                ch = p->data.size - request;
            }
        }
        p = p->next;
    }

    if (q == NULL)
        return ERROR; //没有找到空闲块
    else if (q->data.size == request)
    {
        q->data.state = Busy;
        return OK;
    }
    else
    {
        temp->prior = q->prior;
        temp->next = q;
        temp->data.address = q->data.address;
        q->prior->next = temp;
        q->prior = temp;
        q->data.address += request;
        q->data.size = ch;
        return OK;
    }
    return OK;
}
```



动态回收过程Free_mem ()以及其算法流程图：

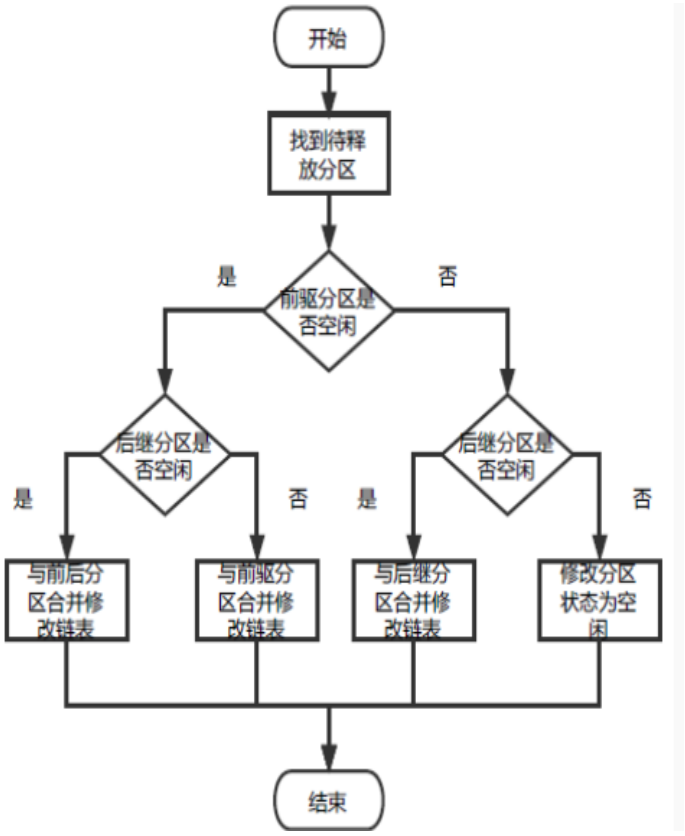
```
//主存回收
Status free_mem(int flag)
{
    DuLNode *p = block_first;
    for (int i = 0; i <= flag; i++)
        if (p != NULL)
            p = p->next;
        else
            return ERROR;

    if (p->data.state == Free)
    {
        printf("该块本为空闲\n");
        return ERROR;
    }
    p->data.state = Free;

    if (p == block_last) //为最后一块
    {
        if (p->prior == block_first)
        {
            p->data.state = Free;
        }
        else
        {
            if (p->prior->data.state == Busy)
            {
                p->data.state = Free;
            }
            else
            {
                p->prior->data.size += p->data.size;
                p->prior->next = p->next;
                p->next->prior = p->prior;
```

```
                p->prior->data.size += p->data.size;
                p->prior->next = p->next;
                p->next->prior = p->prior;
                p = p->prior;
            }
        }
        return OK;
    }
    if (p->prior != block_first && p->prior->data.state == Free) //与前面的空闲块相
    {
        p->prior->data.size += p->data.size;
        p->prior->next = p->next;
        p->next->prior = p->prior;
        p = p->prior;
        return OK;
    }
    if (p->next != block_last && p->next->data.state == Free) //与后面的空闲块相连
    {
        p->data.size += p->next->data.size;
        p->next->next->prior = p;
        p->next = p->next->next;
        return OK;
    }
    if (p->next == block_last && p->next->data.state == Free) //与最后的空闲块相连
    {
        p->data.size += p->next->data.size;
        p->next = NULL;
        return OK;
    }

    printf("\n未知错误！");
    return ERROR;
}
```



六、运行与测试

首次适应算法：

作业 1 申请 130KB

```
请输入需要分配的主存大小(单位:KB): 130
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0               130KB          已分配

  1        130             510KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 2 申请 60KB

```
请输入需要分配的主存大小(单位:KB): 60
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0               130KB          已分配

  1        130             60KB           已分配

  2        190             450KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 3 申请 90KB

```
请输入需要分配的主存大小(单位:KB): 90
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0               130KB          已分配

  1        130             60KB           已分配

  2        190             90KB           已分配

  3        280             360KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 2 释放内存 (60KB)

```
主存分配情况：
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0              130KB          已分配

  1        130            60KB           空闲

  2        190            90KB           已分配

  3        280            360KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 4 申请 200KB

```
主存分配情况：
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0              130KB          已分配

  1        130            60KB           空闲

  2        190            90KB           已分配

  3        280            200KB          已分配

  4        480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 3 释放内存 (90KB)

```
请输入您要释放的分区号: 2

主存分配情况：
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0              130KB          已分配

  1        130            150KB          空闲

  2        280            200KB          已分配

  3        480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 1 释放内存 (130KB)

```
请输入您要释放的分区号: 0

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址        分区大小        状态

  0       0              280KB          空闲

  1       280            200KB          已分配

  2       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 5 申请 140KB

```
主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址        分区大小        状态

  0       0              140KB          已分配

  1       140            140KB          空闲

  2       280            200KB          已分配

  3       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 6 申请 60KB

```
请输入需要分配的主存大小(单位:KB): 60
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址        分区大小        状态

  0       0              140KB          已分配

  1       140            60KB           已分配

  2       200            80KB           空闲

  3       280            200KB          已分配

  4       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 7 申请 50KB

```
主存分配情况：
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址          分区大小          状态

  0        0               140KB           已分配

  1        140             60KB            已分配

  2        200             50KB            已分配

  3        250             30KB            空闲

  4        280             200KB           已分配

  5        480             160KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 6 释放内存 (60KB)

```
主存分配情况：
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址          分区大小          状态

  0        0               140KB           已分配

  1        140             60KB            空闲

  2        200             50KB            已分配

  3        250             30KB            空闲

  4        280             200KB           已分配

  5        480             160KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 8 申请 170KB

```
请输入需要分配的主存大小(单位:KB): 170
内存不足, 分配失败!

主存分配情况：
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址          分区大小          状态

  0        0               140KB           已分配

  1        140             60KB            空闲

  2        200             50KB            已分配

  3        250             30KB            空闲

  4        280             200KB           已分配

  5        480             160KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

最佳适应算法：

作业 1 申请 130KB

```
请输入需要分配的主存大小(单位:KB): 130
分配成功!

主存分配情况:
++++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0              130KB          已分配

  1        130            510KB          空闲

++++++++++++++++++++++++++++++++++++++++++++++++
```

作业 2 申请 60KB

```
请输入需要分配的主存大小(单位:KB): 60
分配成功!

主存分配情况:
++++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0              130KB          已分配

  1        130            60KB           已分配

  2        190            450KB          空闲

++++++++++++++++++++++++++++++++++++++++++++++++
```

作业 3 申请 90KB

```
请输入需要分配的主存大小(单位:KB): 90
分配成功!

主存分配情况:
++++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0        0              130KB          已分配

  1        130            60KB           已分配

  2        190            90KB           已分配

  3        280            360KB          空闲

++++++++++++++++++++++++++++++++++++++++++++++++
```

作业 2 释放内存 (60KB)

```
请输入您要释放的分区号: 1

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            60KB           空闲

  2       190            90KB           已分配

  3       280            360KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 4 申请 200KB

```
1
请输入需要分配的主存大小(单位:KB): 200
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            60KB           空闲

  2       190            90KB           已分配

  3       280            200KB          已分配

  4       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 3 释放内存 (90KB)

```
请输入您要释放的分区号: 2

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            150KB          空闲

  2       280            200KB          已分配

  3       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 1 释放内存 (130KB)

```
请输入您要释放的分区号: 0

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0               280KB          空闲

  1       280             200KB          已分配

  2       480             160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 5 申请 140KB

```
请输入需要分配的主存大小(单位:KB): 140
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0               280KB          空闲

  1       280             200KB          已分配

  2       480             140KB          已分配

  3       620             20KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 6 申请 60KB

```
主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0               60KB           已分配

  1       60              220KB          空闲

  2       280             200KB          已分配

  3       480             140KB          已分配

  4       620             20KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 7 申请 50KB

```
请输入需要分配的主存大小(单位:KB): 50
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              60KB           已分配

  1       60             50KB           已分配

  2       110            170KB          空闲

  3       280            200KB          已分配

  4       480            140KB          已分配

  5       620            20KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 6 释放内存 (60KB)

```
请输入您要释放的分区号: 0

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              60KB           空闲

  1       60             50KB           已分配

  2       110            170KB          空闲

  3       280            200KB          已分配

  4       480            140KB          已分配

  5       620            20KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

作业 8 申请 170KB

```
请输入需要分配的主存大小(单位:KB): 170
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              60KB           空闲

  1       60             50KB           已分配

  2       110            170KB          已分配

  3       280            200KB          已分配

  4       480            140KB          已分配

  5       620            20KB           空闲

+++++++++++++++++++++++++++++++++++++++++++++
```

最差适应算法：

作业 1 申请 130KB

```
请输入需要分配的主存大小(单位:KB): 130
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            510KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 2 申请 60KB

```
请输入需要分配的主存大小(单位:KB): 60
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            60KB           已分配

  2       190            450KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 3 申请 90KB

```
请输入需要分配的主存大小(单位:KB): 90
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            60KB           已分配

  2       190            90KB           已分配

  3       280            360KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 2 释放内存 (60KB)

```
请输入您要释放的分区号: 1

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            60KB           空闲

  2       190            90KB           已分配

  3       280            360KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 4 申请 200KB

```
请输入需要分配的主存大小(单位:KB): 200
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            60KB           空闲

  2       190            90KB           已分配

  3       280            200KB          已分配

  4       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 3 释放内存 (90KB)

```
请输入您要释放的分区号: 2

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              130KB          已分配

  1       130            150KB          空闲

  2       280            200KB          已分配

  3       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 1 释放内存 (130KB)

```
请输入您要释放的分区号: 0

主存分配情况:
++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址        分区大小        状态

  0       0              280KB          空闲

  1       280            200KB          已分配

  2       480            160KB          空闲

++++++++++++++++++++++++++++++++++++++++++++++
```

作业 5 申请 140KB

```
请输入需要分配的主存大小(单位:KB): 140
分配成功!

主存分配情况:
++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址        分区大小        状态

  0       0              140KB          已分配

  1       140            140KB          空闲

  2       280            200KB          已分配

  3       480            160KB          空闲

++++++++++++++++++++++++++++++++++++++++++++++
```

作业 6 申请 60KB

```
请输入需要分配的主存大小(单位:KB): 60
分配成功!

主存分配情况:
++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址        分区大小        状态

  0       0              140KB          已分配

  1       140            140KB          空闲

  2       280            200KB          已分配

  3       480            60KB           已分配

  4       540            100KB          空闲

++++++++++++++++++++++++++++++++++++++++++++++
```

作业 7 申请 50KB

```
请输入需要分配的主存大小(单位:KB): 50
分配成功!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              140KB          已分配

  1       140            50KB           已分配

  2       190            90KB           空闲

  3       280            200KB          已分配

  4       480            60KB           已分配

  5       540            100KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 6 释放内存 (60KB)

```
请输入您要释放的分区号: 4

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              140KB          已分配

  1       140            50KB           已分配

  2       190            90KB           空闲

  3       280            200KB          已分配

  4       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

作业 8 申请 170KB

```
请输入需要分配的主存大小(单位:KB): 170
内存不足, 分配失败!

主存分配情况:
+++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址         分区大小         状态

  0       0              140KB          已分配

  1       140            50KB           已分配

  2       190            90KB           空闲

  3       280            200KB          已分配

  4       480            160KB          空闲

+++++++++++++++++++++++++++++++++++++++++++++++
```

七、改进的方向

1.碎片处理:

内存合并: 作为解决外部碎片最直接的方式，可以实现一个内存紧缩函数。当分配失败或碎片达到一定程度时调用，将所有已分配的块移动到内存一端，合并空闲空间。需要注意实现复杂度和高昂的运行时开销。

碎片阈值调整: 实验中可以研究当分配后剩余空间小于多少时不进行分裂，对外部碎片和内部碎片的影响。

2.数据结构与算法优化:

加速查找: 对于 BF/WF，简单链表查找效率低。可以考虑：

多重链表: 按空闲块大小范围维护多个链表。高级结构: 使用平衡二叉搜索树按大小组织空闲块，可以加速 BF 查找，但插入和删除）操作更复杂。

加速合并: 引入边界标记技术。在每个块的头部和尾部都存储大小和状态信息。回收时，可以通过计算地址直接检查物理相邻块的状态，无需遍历链表即可快速判断是否可以合并，显著提高回收效率。

3.模拟真实性:

考虑更复杂的场景，如进程动态增长内存需求、内存保护等。当前模拟是理想化的，实际操作系统内存管理要复杂得多（。

实验三　请求调页存储管理方式的模拟

一、实验目的

1.理解页式存储管理的基本原理；

2.掌握集中常见的页面淘汰算法。

二、实验内容

1.编程模拟进程内存访问及页面置换算法：

模拟一个具有 320 条指令的作业执行过程。每个页面存放 10 条指令，逻辑地址空间共 32 页。作业最初分配 4 个物理内存块，所有页面均不在内存中。

2.模拟请求分页过程：

3.访问指令在内存：显示物理地址，继续执行。

4.访问指令不在内存（缺页）：记录缺页次数，将页面调入；若内存已满，根据置换算法选择页面淘汰。显示物理地址，继续执行。

5.按照指定规则生成指令执行序列：

随机起点 m ∈ [0, 319]。序列：m, m+1, m1（随机 ∈ [0, m]), m1+1, m2（随机 ∈ [m1+2, 319]), m2+1, 然后重复此模式直到访问 320 条指令。实现 OPT（最佳), FIFO（先进先出), LRU（最近最少使用）三种页面置换算法。

6.统计与对比分析：

使用上述模拟流程，分别统计 OPT, FIFO, LRU 算法在分配 4 个物理块时的缺页率，并进行对比分析。

7.改变物理块数的影响分析：

将分配的物理块数改为 8 块，重复步骤 2，统计三种算法的缺页率。

分析物理块数增加对缺页率的影响规律。

三、实现思路

1.数据结构设计： 设计页表结构、物理内存块表示、指令访问序列存储。

2.指令序列生成： 实现一个函数，根据实验要求（随机起点，顺序执行，向前/向后跳转）循环生成总共 320 条指令的访问地址序列。

3.地址转换与缺页判断： 实现一个核心函数，输入逻辑地址（指令序号），计算出页号和页内偏移。根据页号查询页表，判断页面是否在内存。

4.缺页处理：如果缺页，增加缺页计数。检查物理内存是否已满。若未满，找一个空闲块分配，更新页表和内存状态。若已满，调用相应的页面置换算法选择要淘汰的页面。更新页表、内存状态。

5.页面置换算法实现：

FIFO: 使用队列维护内存中页面的进入顺序。淘汰队头页面。

LRU: 需要记录页面最近一次被访问的时间或访问次序。淘汰最长时间未被访问的页面。可以使用时间戳、计数器或列表维护。

OPT: 需要扫描 后续 的指令访问序列。对于内存中的每个页面，找到其下一次被访问的位置。淘汰下一次访问距离当前最远的页面。这通常作为性能基准，因为它需要预知未来。

6.模拟循环： 遍历生成的 320 条指令访问序列，对每条指令调用地址转换和缺页处理函数。

7.统计与输出： 在模拟结束后，计算总指令访问次数和总缺页次数，得出缺页率（缺页次数 / 总访问次数）。输出结果。

8.多次运行与比较： 分别设置不同的物理块数（4 和 8），以及不同的置换算法（OPT, FIFO, LRU），运行模拟程序，记录并比较缺页率。

三、主要的数据结构

1. 常量定义

```
#define maxn 320      //序列个数
#define max (maxn + 20) //数组大小
#define maxp (max / 10) //最大页数
```

2. 全局数组

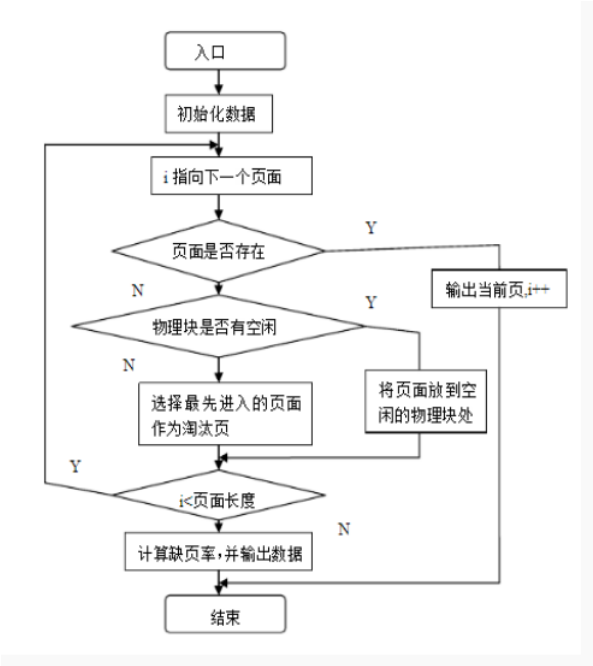
```
int inst[max];    //指令序列：存储随机生成的320条指令
int page[max];    //页地址流：由指令序列转换而来的页号
int in[maxp];     //该页是否在内存里的标记数组，提高查找效率
int pin[maxp];    //现在在内存里的页的数组，存储当前内存中的页号
```

3. 全局变量

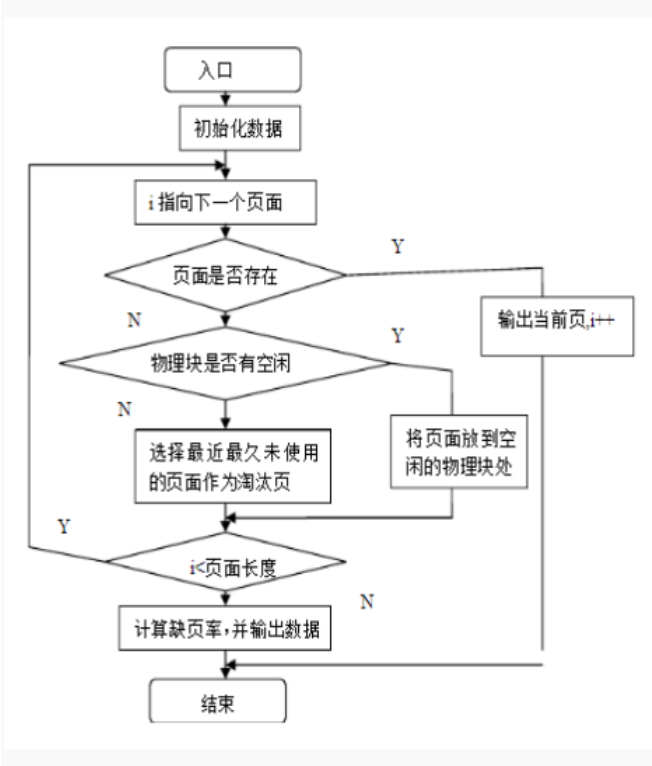
int size;　　//内存能容纳的页数（可由用户设置，通常是 4 到 32 之间）

五、算法流程图

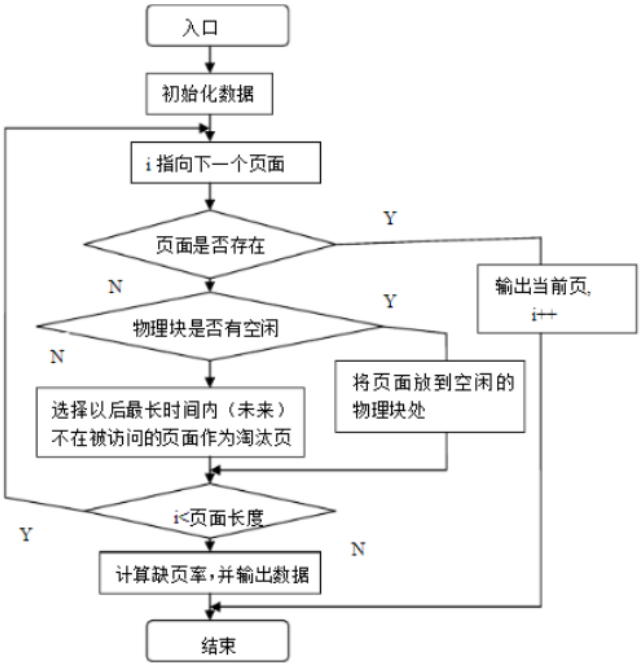
一、先来先服务（FIFO）



二．最近最少使用法（LRU）



三．最佳页面替换法（OPT）



七、运行与测试

首先是先初始化和定义页框的数量：

```
1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  1

New page address sequence is set OK!!!

1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  2

Please input the size of memory page number:  4
```

下面是使用 FIFO 算法：

```
1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  3


By FIFO algorithm, the fault-page number is:  281
                   the hit ratio is :  0.12
```

然后是 LRU 算法：

```
1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  4


By LRU algorithm, the fault-page number is:  278
                   the hit ratio is :  0.13
```

最后是 OPT 算法：

```
1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  5


By OPT algorithm, the fault-page number is:  212
                   the hit ratio is :  0.34
```

八、改进的方向

1. 需要更真实的指令访问模式： 虽然 produce_inst() 函数试图模拟局部性，但它与实验要求的具体生成模式可能不完全一致。仔细对比并调整 produce_inst() 函数，确保它精确实现了这个模式，因为指令序列的局部性特征会直接影响缺页率。

2. 实验内容提到“访问指令在内存：显示物理地址，继续执行。”当前代码只

关注缺页和命中率，并没有实际输出每条指令对应的物理地址。我觉得：在 main 函数的循环中，每次处理一个 page[i] 时，如果该页在内存中，计算并打印出对应的物理地址。这需要你在 pin 数组中记录页号所在的物理块号。例如，如果 pin[j] = page[i]，那么页号 page[i] 就在物理块 j 中。物理地址可以简单地表示为 物理块号 * 10 + 页内偏移。

3. 目前只统计了缺页次数和命中率。可以增加更多有价值的指标，并考虑用图表进行可视化。我觉得可以添加平均访问时间：假设每次命中花费 t_hit 时间，每次缺页花费 t_miss 时间（通常 t_miss 远大于 t_hit）。你可以计算总的模拟时间，并将其作为性能指标。或者是内存利用率： 简单展示在整个模拟过程中，物理内存块被占用的平均比例。

实验四 简单文件系统的模拟实现

一、实验目的

1.了解文件系统的概念，熟悉文件系统的功能；

2.通过模拟实验掌握文件系统对与文件的创建、删除、打开、关闭、读和写等基本操作进行处理的。

二、实验内容

（1）假设文件系统使用树形结构目录进行文件管理，支持多级目录结构和小数据文件，开发程序实现对目录和文件操作的模拟功能。

（2）为简单起见，对文件系统中所有文件的 FCB 和目录项进行融合，直接使用链表结构进行管理。不使用单独的索引结点，文件数据也不作离散和分开存储，使用连续存储形式直接存储在 FCB 中。每个目录项必须包括文件名信息，可以根据功能需要自行扩展。

（3）要求必须实现如下文件操作：

- dir：查看当前目录下的文件。
- read：读文件数据。
- write：写入文件数据。
- delete：删除文件。
- rm：删除目录。
- cd：更改录前工作目录。
- mkdir：在当前目录下创建目录。
- creat：在当前目录下创建文件。

（4）可选实现如下相关操作：

- open：打开文件。
- close：关闭文件。

（5）可选实现文件系统的持久化。

三、实现思路

1.核心数据结构：使用一个名为 `filenode` 的结构体来统一表示文件和目录。该结构体包含文件名、类型标识、文件内容、指向父节点的指针、指向第一个子节点的指针、以及指向同一层级中前一个和后一个节点的指针。这种设计允许构建一个多叉树结构，其中每个目录节点可以有多个子节点（文件或其他目录），这些子节点通过 `child` 指针连接到父目录，并通过 `next` 和 `prev` 指针形成一个双向链表。

2.文件系统初始化：程序启动时，调用 `createroot()` 函数创建一个根目录。此根目录作为所有文件和目录的顶层祖先。全局指针 `root` 指向根节点，`recent` 指向当前工作目录，初始时也指向根目录。全局变量 `path` 存储当前工作目录的路径字符串。

3.命令解析与分发：`main` 函数中有一个主循环，每次循环调用 `run()` 函数。`run()` 函数首先打印当前路径提示符，然后读取用户输入的命令。通过 `strcmp()` 函数比较用户输入的命令与预设的命令字符串，然后调用相应的处理函数。

4.基本操作实现：

- **创建目录 / 创建文件**
  - 检查当前目录下是否已存在同名同类型的文件/目录。
  - 如果当前目录没有子节点，则新节点成为第一个子节点。
  - 如果已有子节点，则新节点被添加到子节点链表的末尾。新创建的目录节点的 `parent` 指针指向 `recent`，或者为 `NULL`。
- **切换目录（cd）:**
  - 处理特殊路径 `"."` 和 `".."` 。对于 `".."`，通过 `recent->parent` 向上移动并相应修改 `path` 字符串。
  - 对于其他路径，调用 `findpath()` 函数。
- **路径查找 ：**
  - 该函数负责解析路径字符串，并在文件系统树中查找目标目录。

- 支持绝对路径和相对路径。
- 逐级解析路径中的目录名，在每级目录的子节点链表中查找匹配的目录名。
- 如果找到，更新 `recent` 指针到目标目录，并更新 `path` 字符串。
- 处理路径错误。

- **显示目录内容 ：**

  - 遍历当前目录 的子节点链表。
  - 打印每个子节点的类型和名称。
  - 统计并显示子目录和文件的数量。也显示 `"."` 和 `".."`。

- **读文件 / 写文件 ：**

  - 在当前目录的子节点链表中查找指定名称的文件。
  - `read`：如果找到文件，则打印其 `content` 成员。
  - `write`：如果找到文件，则读取用户输入并存入其 `content` 成员。

- **删除文件/ 删除目录 ：**

  - 在当前目录的子节点链表中查找指定名称的文件/目录。
  - 找到后，从链表中移除该节点（修改其前驱节点的 `next` 指针和后继节点的 `prev` 指针，或者修改父节点的 `child` 指针），然后使用 `free()` 释放节点内存。
  - **注意**：当前 `rm` 实现并不支持递归删除非空目录。它仅删除指定的目录节点本身。若要删除非空目录，需要额外实现递归删除其所有子内容的逻辑。

四、主要的数据结构

```
struct filenode
{
    char filename[FILENAME_LENGTH];
    int isdir;
    char content[255];
    struct filenode *parent;
    struct filenode *child;
    struct filenode *prev;
    struct filenode *next;
};
```

- `filename`：存储文件或目录的名称，长度受 `FILENAME_LENGTH` 宏限制。
- `isdir`：整数类型，作为标志位。如果为 `1`，表示该节点是一个目录；如果为 `0`，表示是一个文件。

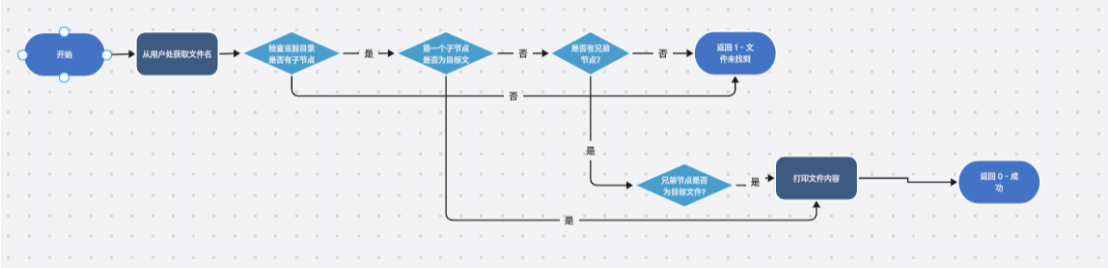
- content：字符数组，用于存储文件的内容。对于目录节点，此字段未使用。
- parent：指向其父目录节点的指针。根节点的 parent 为 NULL。在 mkdir 和 create 的实现中，如果一个节点不是其父节点的第一个孩子，它的 parent 指针被设为 NULL，这是一种简化的处理方式，依赖于通过 child 和 next 形成的链表进行遍历。
- child：指向该目录的第一个子节点的指针。如果该节点是文件，或者是一个空目录，则 child 为 NULL。
- prev：指向同一父目录下的前一个兄弟节点的指针。父节点的第一个子节点的 prev 为 NULL。
- next：指向同一父目录下的后一个兄弟节点的指针。父节点的最后一个子节点的 next 为 NULL。

```
struct filenode *root, *recent, *temp, *ttemp, *temp_child;
char path[PATH_LENGTH], command[COMMAND_LENGTH], temppath[PATH_LENGTH], recentpath[PATH_LENGTH];
```

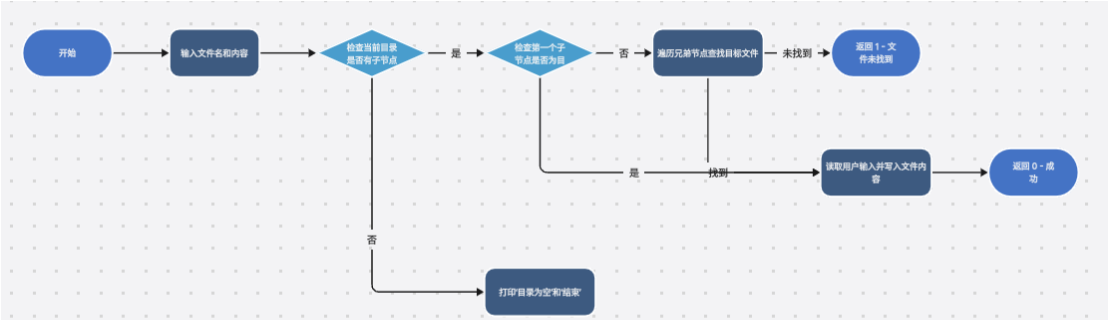
- struct filenode *root：指向文件系统的根目录节点。
- struct filenode *recent：指向当前活动目录（工作目录）的节点。用户的所有操作都是相对于这个目录进行的。
- struct filenode *temp, *ttemp, *temp_child：临时指针，用于各种操作中遍历或创建节点。
- char path[PATH_LENGTH]：存储当前工作目录的绝对路径字符串（例如，"/usr/bin"）。
- char command[COMMAND_LENGTH]：存储用户输入的命令。
- char temppath[PATH_LENGTH]：临时存储路径字符串，主要在 findpath 中用于路径回溯。
- char recentpath[PATH_LENGTH]：在 findpath 中用于存储路径的单个组成部分。

五、算法流程图

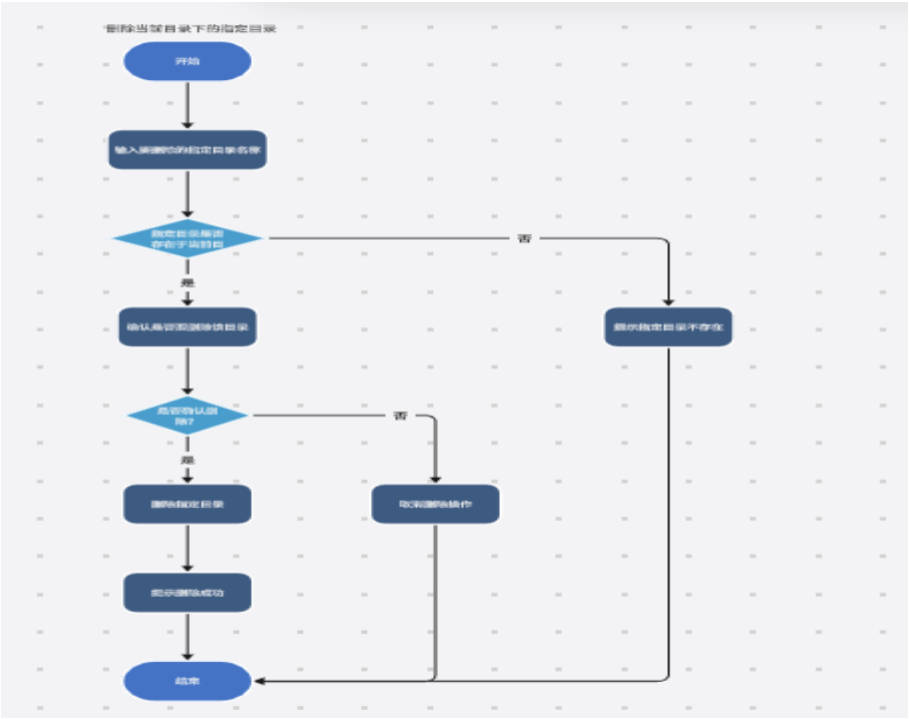
函数 `int read()`：



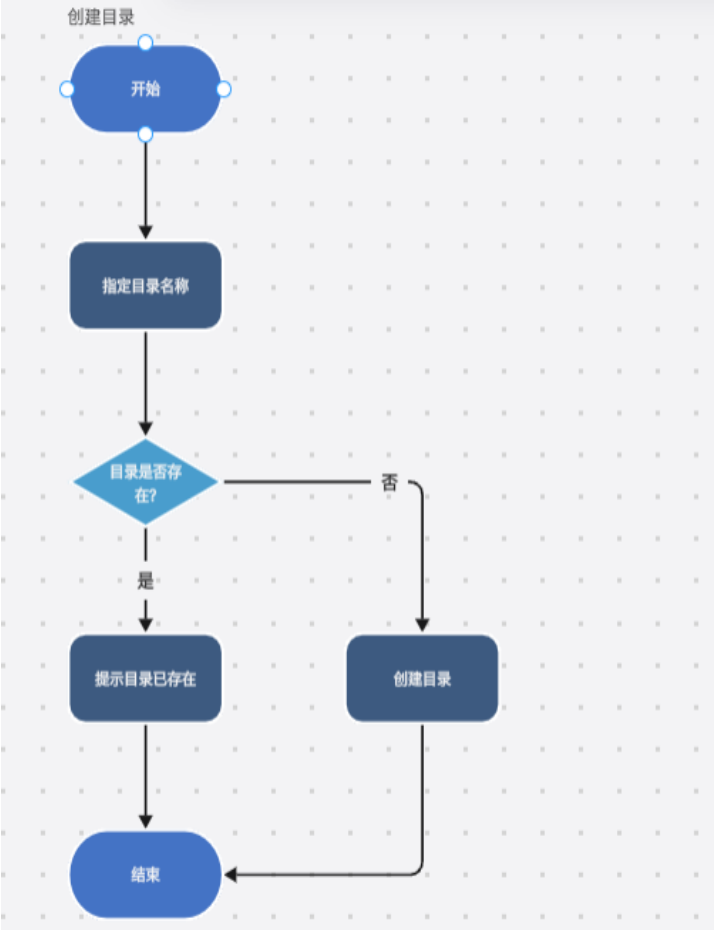
函数 `int write()`：



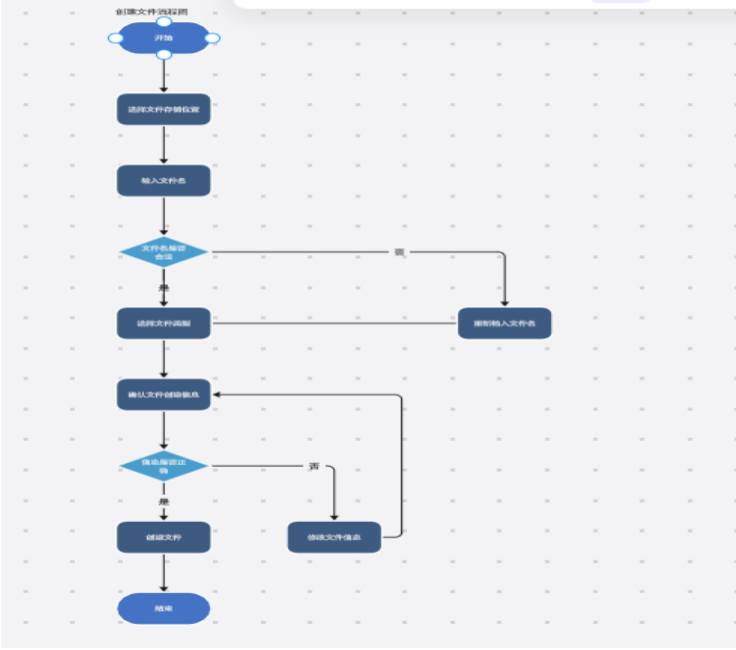
函数 `int del()`:



函数 `int mkdir()`:



函数 `int create()`:



六、运行与测试

启动程序:

```
**********************************************************************
**********************操作系统课程设计项目************************
* 简单文件系统模拟                                          *
* 键入help可以获取帮助                                        *
**********************************************************************
**********************************************************************

filesystem:/>
```

测试 mkdir, dir, cd:

```
filesystem:/>mkdir docs
目录建立成功!
filesystem:/>mkdir pics
目录建立成功!
filesystem:/>dir
      <DIR>                       .
      <DIR>                       docs
      <DIR>                       pics
Total: 3 directors      0 files
filesystem:/>cd docs
filesystem:/docs>mkdir reports
目录建立成功!
filesystem:/docs>dir
      <DIR>                       ..
      <DIR>                       reports
Total: 2 directors      0 files
filesystem:/docs>cd reports
filesystem:/docs/reports>dir
      <DIR>                       ..
Total: 1 directors      0 files
filesystem:/docs/reports>cd ..
filesystem:/docs>cd ..
filesystem:/>dir
      <DIR>                       .
      <DIR>                       docs
      <DIR>                       pics
Total: 3 directors      0 files
```

测试 create, write, read:

```
filesystem:/>cd docs
filesystem:/docs>create notes.txt
文件创建成功!
filesystem:/docs>dir
      <DIR>                       ..
      <DIR>                       reports
      <FILE>                      notes.txt
Total: 2 directors      1 files
filesystem:/docs>write notes.txt
This is a test note.
文件写入成功!
filesystem:/docs>read notes.txt
This is a test note.
filesystem:/docs>
```

测试 delete:

```
filesystem:/docs>delete notes.txt
文件已删除!
filesystem:/docs>dir
      <DIR>                        ..
      <DIR>                        reports
Total: 2 directors      0 files
filesystem:/docs>delete non_existent_file.txt
不存在该文件!
```

测试 rm **(空目录)**:

```
filesystem:/docs>rm reports
目录已删除!
filesystem:/docs>dir
      <DIR>                       ..
Total: 1 directors      0 files
filesystem:/docs>cd ..
filesystem:/>rm docs
目录已删除!
filesystem:/>dir
      <DIR>                       .
      <DIR>                       pics
Total: 2 directors      0 files
```

测试 rm **(非空目录 - 根据当前代码，只会删除目录节点本身)**:

```
filesystem:/>mkdir testdir
目录建立成功!
filesystem:/>cd testdir
filesystem:/testdir>create file_in_testdir.txt
文件创建成功!
filesystem:/testdir>cd ..
filesystem:/>
```

```
filesystem:/>rm testdir
目录已删除!
filesystem:/>dir
      <DIR>                    .
      <DIR>                    pics
Total: 2 directors      0 files
```

目录 testdir 的节点被删除。其子文件 file_in_testdir.txt 会成为孤儿节点，内存未释放，且无法再通过文件系统访问。这表明当前 rm 不处理非空目录的子内容。

测试重名创建:

```
filesystem:/>mkdir myfolder
目录建立成功!
filesystem:/>mkdir myfolder
目录已存在!
filesystem:/>create afile.txt
文件创建成功!
filesystem:/>create afile.txt
文件已存在!
```

七、改进的方向

1.持久化存储:当前文件系统完全存在于内存中，程序退出后所有数据丢失。可以实现将文件系统结构序列化到磁盘文件，并在程序启动时加载回来。

2.rm 处理非空目录:当前 `rm` 命令不能递归删除非空目录的内容。应修改 `rm` 函数，使其在删除一个目录前，先递归删除其所有子文件和子目录。

3.文件/目录元数据扩展:`filenode` 结构可以扩展以包含更多元数据，如文件大小、创建时间、修改时间、权限等。

4.实现 `open` 和 `close` 命令:实现可选的 `open` 和 `close` 功能。这可能涉及到维护一个“打开文件表”，并为打开的文件分配某种形式的句柄或描述符。`read` 和 `write` 将针对打开的文件进行操作。

5.数据结构与算法效率:对于非常大的目录，线性查找效率较低。可以考虑为每个目录的子节点使用更高效的查找结构，如哈希表或平衡二叉搜索树。