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 23 级人工智能 1 班

学　　号 3123009048

学生姓名廖卓远

指导教师李剑锋、苏畅

2025 年 5 月 28 日

实验一 银行家算法与避免死锁

一、实验目的

1.掌握死锁产生的必要条件及银行家算法的核心原理。

2.通过编程实现银行家算法，模拟资源分配与安全性检查流 程。

二、实验内容

1.模拟一个银行家算法：设置数据结构设计安全性算法

2.初始化时让系统拥有一定的资源

3.用键盘输入的方式申请资源

4.如果预分配后，系统处于安全状态，则修改系统的资源分配 情况

5.如果预分配后，系统处于不安全状态，则提示不能满足请求

三、实现思路

1.实验原理

进程申请资源时，系统通过一定的算法判断本次申请是否不可能产生死锁（处于安全状态）。若可能产生死锁（处于不安全状态），则暂不进行本次资源分配，以避免死锁。算法有著名的银行家算法。

(1). 什么是系统的安全状态和不安全状态？

所谓安全状态，是指如果系统中存在某种进程序列＜P1，P2，…，Pn＞，系统按该序列为每个进程分配其所需要的资源，直至最大需求，则最终能使每个进程都可顺利完成，称该进程序列＜P1，P2，…，Pn，＞为安全序列。

如果不存在这样的安全序列，则称系统处于不安全状态。

(2). 银行家算法

把操作系统看作是银行家，操作系统管理的资源相当于银行家管理的资金，进程向操作系统请求分配资源相当于用户向银行家贷款。

为保证资金的安全，银行家规定:

①当一个顾客对资金的最大需求量不超过银行家现有的资金时就可接纳该顾客;

②顾客可以分期贷款，但贷款的总数不能超过最大需求量;

③当银行家现有的资金不能满足顾客尚需的贷款数额时,对顾客的贷款可推迟支付,但总能使顾客在有限的时间里得到贷款;

④当顾客得到所需的全部资金后,一定能在有限的时间里归还所有的资金。

操作系统按照银行家制定的规则设计的银行家算法为：

（1）进程首次申请资源的分配：如果系统现存资源可以满足该进程的最大需求量，则按当前的申请量分配资源，否则推迟分配。

（2）进程在执行中继续申请资源的分配：若该进程已占用的资源与本次申请的资源之和不超过对资源的最大需求量,且现存资源能满足该进程尚需的最大资源量，则按当前申请量分配资源，否则推迟分配。

（3）至少一个进程能完成：在任何时刻保证至少有一个进程能得到所需的全部资源而执行到结束。

2. 设计方案

银行家算法 bank()函数：

Requesti：进程 Pi 的请求向量。 0<=j<=m-1

1.若 Requesti[j]≤Need[i,j]，转向 2，否则出错。

2.若 Requesti[j]≤Available[j]，转向 3，否则等待。

3.系统试探着把资源分配给进程 Pi，修改下面内容：

Available[j]=Available[j]–Requesti[j];

Allocation[i,j]=Allocation[i,j]+Requesti[j]; Need[i,j]=Need[i,j]–Requesti[j];

4.试分配后,执行安全性算法,检查此次分配后系统是否处于安全状态。若安全,才正式分配；否则，此次试探性分配作废，进程 Pi 等待。

安全性算法 safe()函数：

1.初始化：设置两个向量 Work(1×m)和 Finish(1×n) Work——系统可提供给进程继续运行所需各类资源数，初态赋值 Available Finish——系统是否有足够资源分配给进程，初值 false.

2.从进程集合中满足下面条件进程： Finish[i] = false; Need[i,j] ≤ Work[j]; 若找到，执行 3，否则，执行 4。

3.进程 Pi 获得资源，可顺利执行，完成释放所分配的资源。

Work[j] = Work[j]+Allocation[i,j]; Finish[i] = true; go to 2.

4.若所有进程 Finish[i] = true，表示系统处于安全状态，否则处于不安全状态。

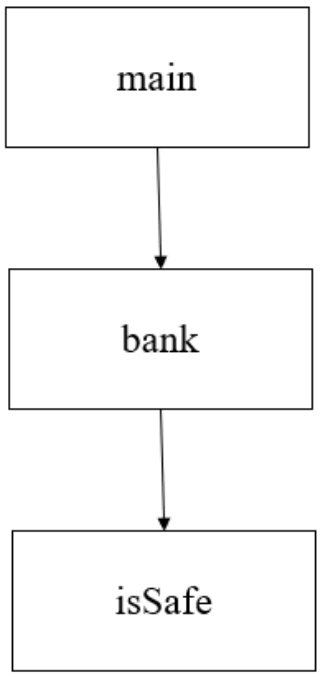
总而言之，先对用户提出的请求进行合法性检查，即检查请求的是否不大于需要的，是否不大于可利用的。若请求合法，则进行试分配。最后对试分配后的状态调用安全性检查算法进行安全性检查。若安全，则分配，否则，不分配，恢复原来状态，拒绝申请。

四、主要的数据结构

可利用资源向量 int Available[m] m 为资源种类

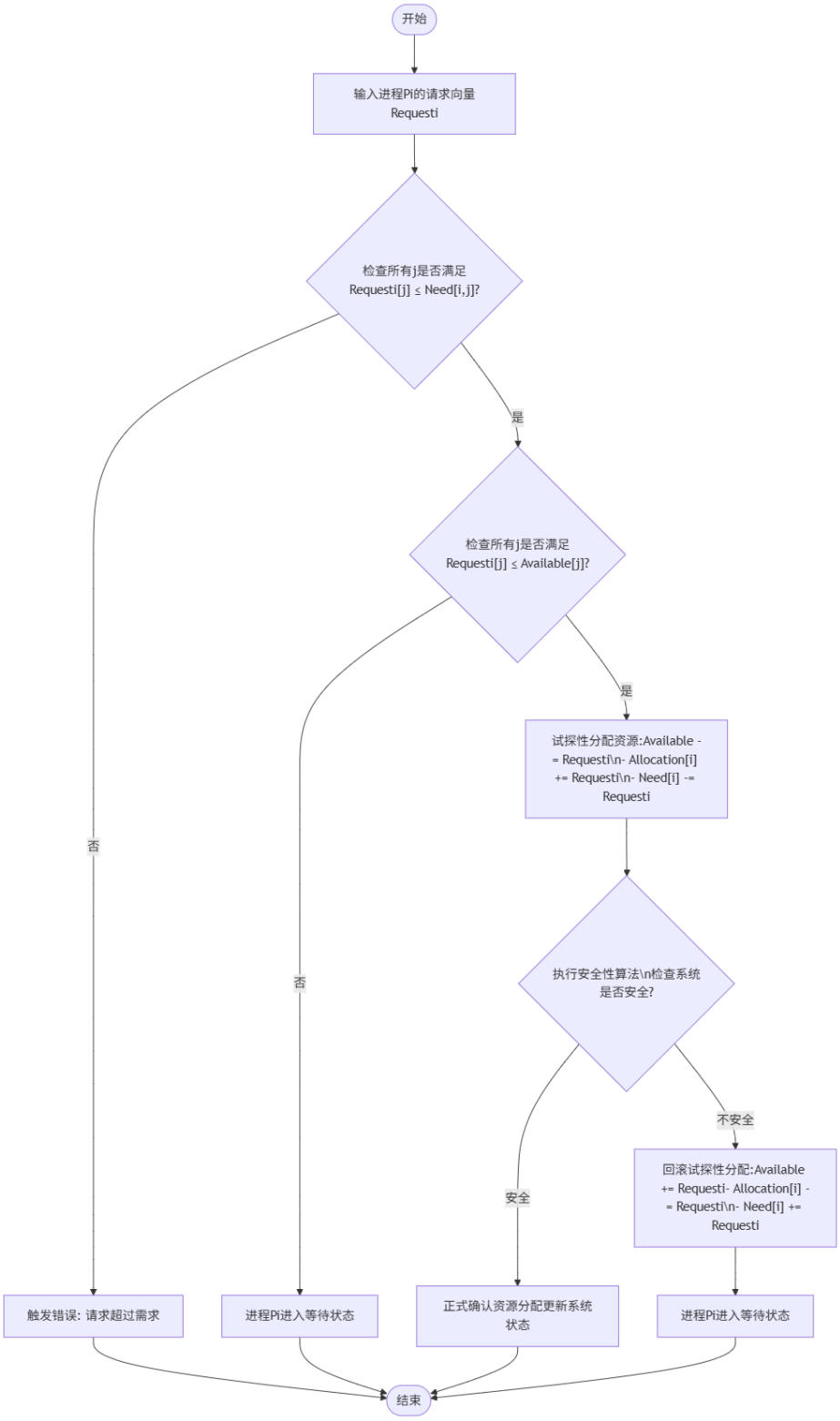
最大需求矩阵	int Max[n][m]　n 为进程的数量
分配矩阵	int Allocation[n][m]
还需资源矩阵	int need[i][j]=Max[i][j]-Allocation[i][j]
申请资源数量	int Request [m]
工作向量	int Work[m] int Finish[n]

各函数调用关系

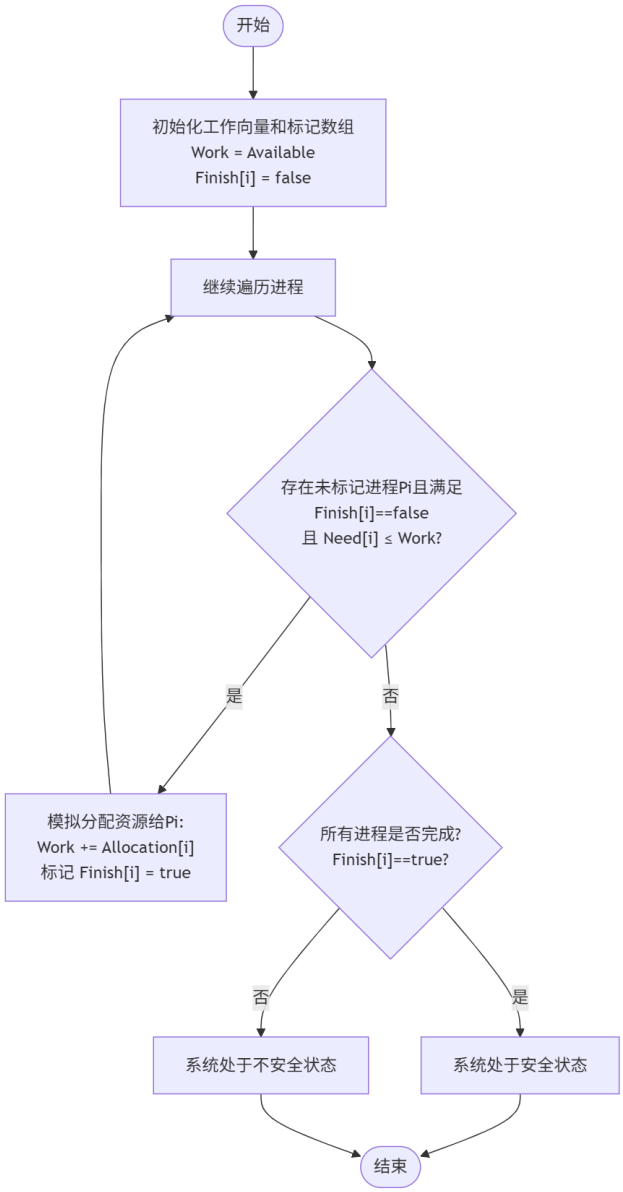


五、算法流程图

银行家算法 bank()流程图：


开始
输入进程Pi的请求向量 Requesti
检查所有j是否满足 Requesti[j] ≤ Need[i,j]?
否
是
检查所有j是否满足 Requesti[j] ≤ Available[j]?
否
是
试探性分配资源:Available -= Requesti\n- Allocation[i] += Requesti\n- Need[i] -= Requesti
执行安全性算法\n检查系统是否安全?
安全
不安全
回滚试探性分配:Available += Requesti- Allocation[i] -= Requesti\n- Need[i] += Requesti
触发错误: 请求超过需求
进程Pi进入等待状态
正式确认资源分配更新系统状态
进程Pi进入等待状态
结束

safe()函数流程图：



六、运行与测试

```
T0时刻安全
----- T1时刻P1请求资源(1,0,2) -----
资源已分配

----- T2时刻P4请求资源(3,3,0) -----
进程4需要等待

----- T3时刻P0请求资源(0,2,0) -----
进程0需要等待（分配会导致不安全状态）
```

七、改进的方向

没有做好分配资源分配和时刻安全的接口，数值里是直接在全局函数里修改的，导致用别的例子跑这段代码会比较难。由于程序是自己写的没有老师提供的

模板，代码的注释较少，可读性较差，下次应该更加注重写代码的规范性。isSafe函数有三重循环，时间复杂度较高，也许还可以进行时间复杂度的优化。

实验二 动态分区分配方式的模拟

一、实验目的

理解动态分区分配方式中使用的数据结构和分配算法，并编程模 拟分配和释放流程，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. 用 C 语言分别实现采用首次适应算法和最佳适应算法的动态分区分配过程和回收过程。其中，空闲分区通过空闲分区链（表）来管理；在进行 内存分配时，系统优先使用空闲区低端的空间。

2. 假设初始状态下，可用的内存空间为 640KB，并有下列的请求序列：

•作业 1 申请 130KB

•作业 2 申请 60KB

•作业 3 申请 100KB

•作业 2 释放 60KB

•作业 4 申请 200KB

•作业 3 释放 100KB

•作业 1 释放 130KB

•作业 5 申请 140KB

•作业 6 申请 60KB

•作业 7 申请 50KB

•作业 8 申请 60KB

请分别采用首次适应算法和最佳适应算法进行内存的分配和回收，要求每次分配和回收后显示出空闲内存分区链的情况。

三、实现思路

1.实验原理

采用可变分区存储管理，选用首次适应算法实现主存分配和回收。

FIFO（首次适应算法）：从空闲分区表的第一个表目起查找该表，把最先能

够满足要求的空闲区分配给作业，这种方法目的在于减少查找时间。为适应这种算法，空闲分区表（空闲区链）中的空闲分区要按地址由低到高进行排序。该算法 优先使用低址部分空闲区，在低址空间造成许多小的空闲区，在高地址空间保留大的空闲区。BF（最佳适应算法）：所谓“最佳”是指，每次为作业分配内存时，总是把能满足要求，又是最小的空闲分区分配给作业，避免“大材小用”。为了加速寻找，该算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。这样，第一次找到的能满足要求的空闲区必然是最佳的。孤立地看，最佳适应算法 似乎是最佳的，然后在宏观上却不一定。因为每次分配后所切割下来的剩余部分总是最小的，这样，在存储器中会留下许多难以利用的碎片。

2.设计方案

①主存空间分配过程

当有进程要求分配 XKB 主存时，按照进程的需要量 XKB，从链头开始找到第一个空闲区；

A、若空闲区小于 XKB，则沿链找下一个空闲区；

B、若空闲区等于 XKB，则该空闲区全部分给作业，并将空闲区从链中摘除，即修改下一个空闲区的向后指针为该空闲区的后向指针。修改上一个空闲区的前向指针为该空闲区的前向指针；

C、若空闲区大于 XKB，则空闲区大小=空闲区大小-XKB，修改下一个空闲区的向后指针(后向指针+XKB)。修改上一个空闲区的前向指针为(前向指针+XKB)；

D、登记已分配区表，输出系统中各数据结构的值，返回分配给作业的主存始址。

②主空空间回收过程

当一个作业执行完成撤离时，作业所占的分区应该归还给系统，归还的分区如果与其它空闲区相邻，则应合成一个较大的空闲区，登记在空闲区说明表中。例如，在上述中列举的情况下，如果作业 2 撤离，归还所占主存区域时，应与上、下相邻的空闲区一起合成一个大的空闲区登记在空闲区说明表中。

回收作业占用的内存总共有四种情况,其中三种情况如下:

A、查到有与释放区地址邻接的空闲区，如果是下相邻，则在空闲区链中找到正确的插入位置：把下邻空闲区的前向指针内容送缓冲区的前向指针，修改下邻

空闲区的前向指针指向释放区始址，把上邻空闲区的后向指针内容送释放区的后向指针，修改上邻空闲区的后向指针指向释放区始址。

B、如果是上相邻，则在空闲区链中找到正确的插入位置：把上邻空闲区的前向指针内容送缓冲区的前向指针，修改上邻空闲区的前向指针指向释放区始址，把下邻空闲区的后向指针内容送释放区的后向指针，修改下邻空闲区的后向指针指向释放区始址。

C、查到有与释放区地址邻接的空闲区，如果是下相邻，则在空闲区链中找到正确的插入位置：把下邻空闲区的前向指针内容送缓冲区的前向指针，修改下邻空闲区的前向指针指向释放区始址，把上邻空闲区的后向指针内容送释放区的后向指针，修改上邻空闲区的后向指针指向释放区始址。

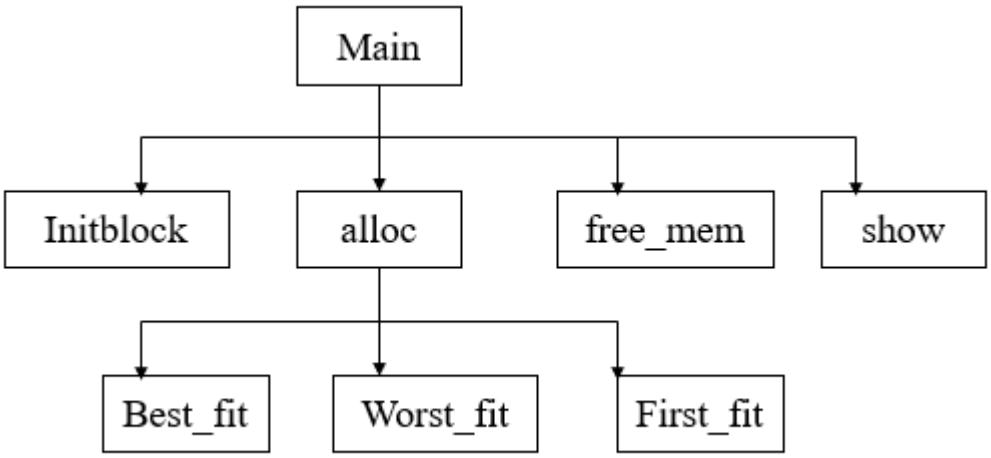
D、如果是上相邻，则在空闲区链中找到正确的插入位置：把上邻空闲区的前向指针内容送缓冲区的前向指针，修改上邻空闲区的前向指针指向释放区始址，把下邻空闲区的后向指针内容送释放区的后向指针，修改下邻空闲区的后向指针指向释放区始址。

四、主要的数据结构

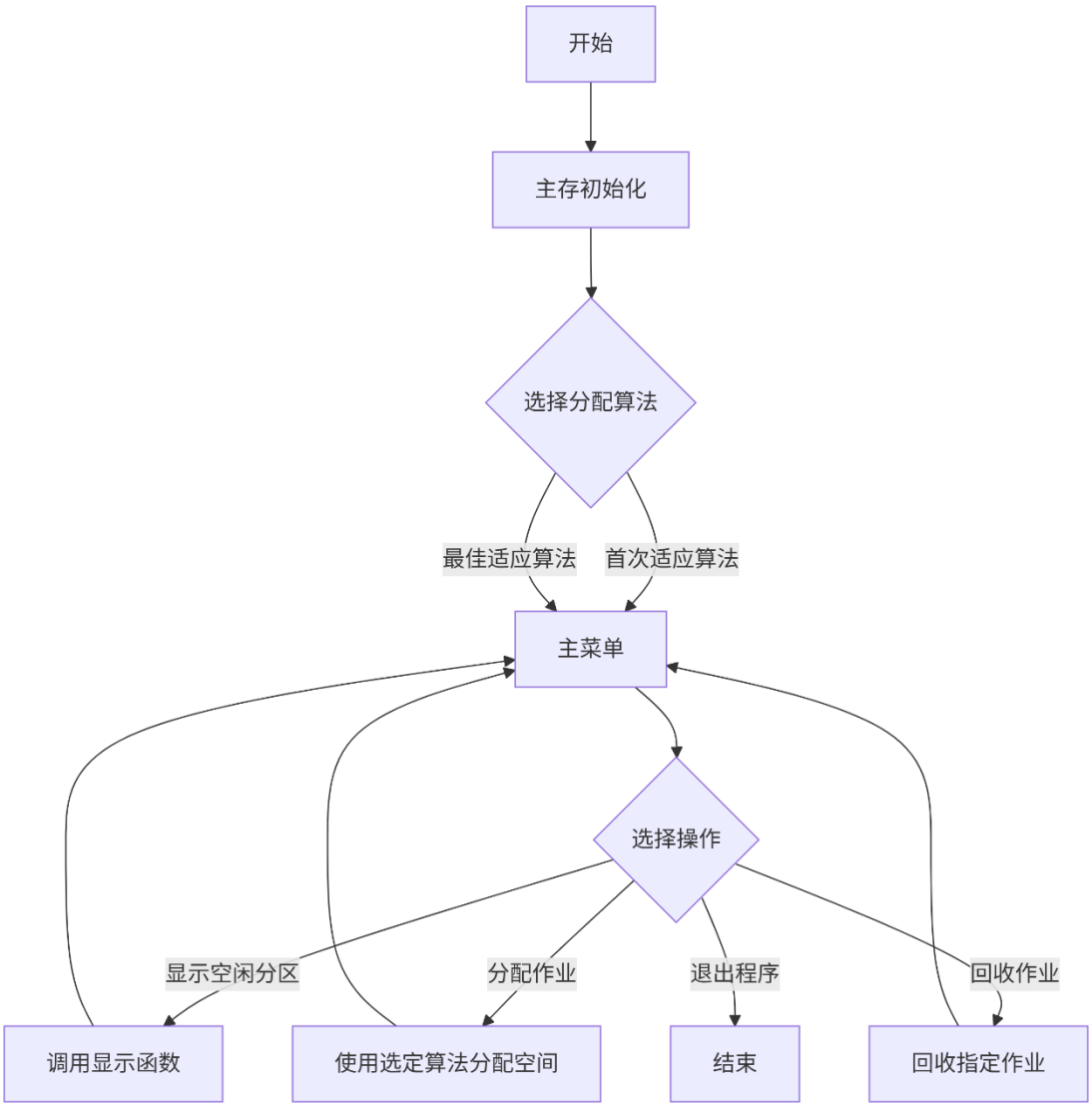
```
#define Free 0
#define Busy 1
#define OK 1
#define ERROR 0
#define MAX_length 640

typedef int Status;
int flag;

typedef struct freearea
{
    long size;
    long address;
    int state;
} ElemType;
```

```
typedef struct DuLNode
{
    ElemType data;
    struct DuLNode *prior;
    struct DuLNode *next;
} DuLNode, *DuLinkList;

DuLinkList block_first;
DuLinkList block_last;
```

各函数调用关系



五、算法流程图



六、运行与测试

Test Pass | 步骤得分：**100** | 2025-04-29 16:02:14

当前步骤自动评测得分（百分制）： 100

> — 当前分数是百分制，最后转换为实际得分。

Congratulations on getting 100 points!!!

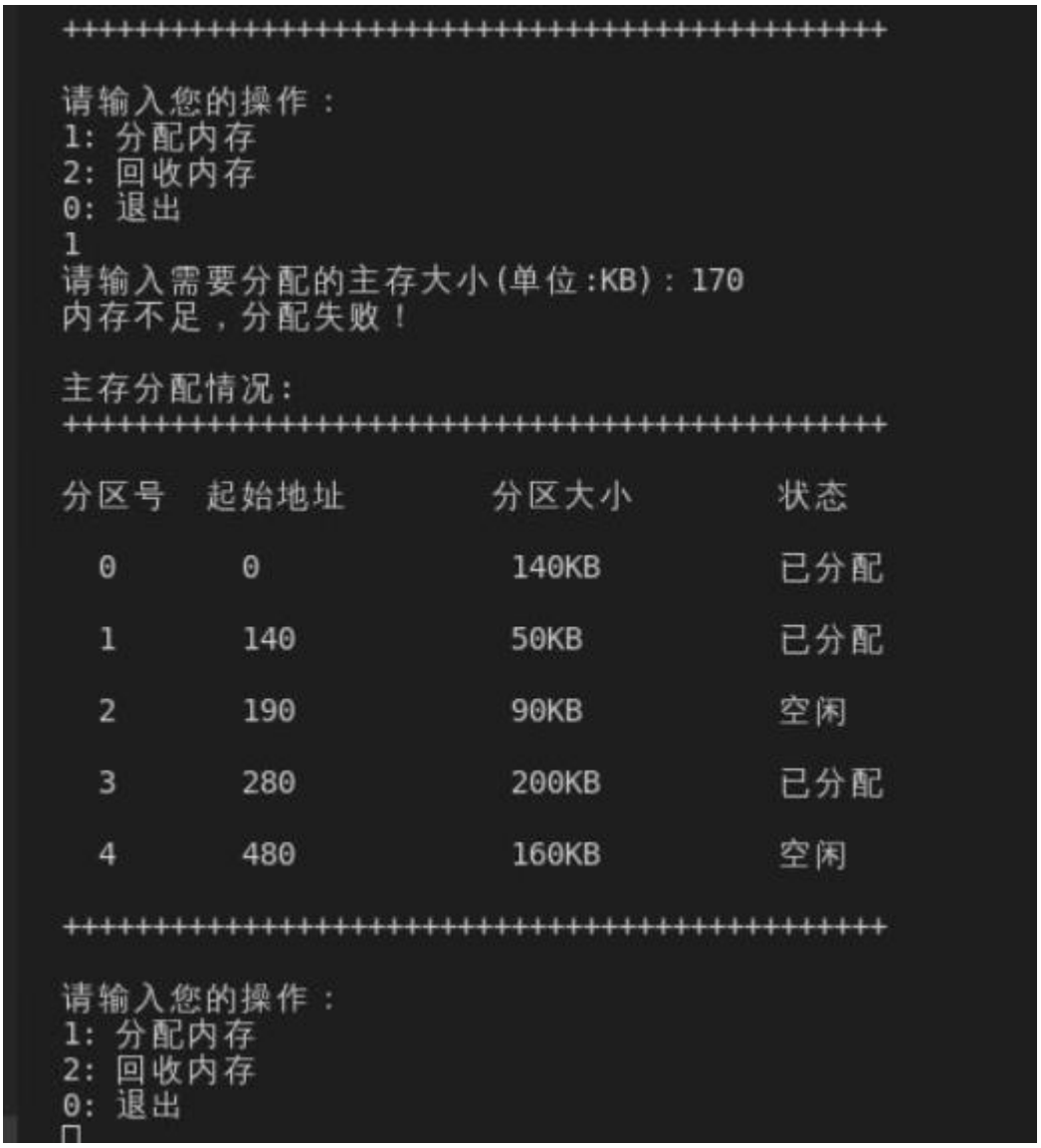
1. 首次适应方法：

```
请输入您的操作：
1: 分配内存
2: 回收内存
0: 退出
1
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

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

2. 最佳适应算法：

```
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

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

3. 最坏适应算法：



七、改进的方向

可以尝试使用更高效的数据结构管理空闲分区，例如,分离空闲链表，加快最佳适应算法的查找速度或红黑树，平衡二叉树，这样可以加速空闲分区的插入、删除和查找操作。测试的案例其实还不够多，感觉还可以添加一些案例。同时显示不太直观，可以看到输出显示里面没有写分区号对应作业几，导致显示不太直观。

思考

（1） 请结合理论和本次模拟数据对比分析采用上述不同算法对内存的分配和回收速度有什么不同的影响？

1. 答：

首次适应算法：

分配速度：最快。从空闲链表的头部开始查找，找到第一个满足大小的分区即分配，无需遍历整个链表。

回收速度：检查前后分区是否空闲，时间复杂度为 $O(1)$。

最佳适应算法：

分配速度：最慢。需遍历整个链表以找到最小的满足要求的分区，时间复杂度为 $O(n)$。

回收速度：与首次适应类似，合并操作的开销相同，但分配时的碎片可能更多，导致后续需更长时间。

最坏适应算法：

分配速度：较慢。需遍历链表找到最大的空闲分区，时间复杂度为 $O(n)$，但是对于内存利用率较低。

回收速度：与首次适应相同。

（2）请结合理论和本次模拟数据对比分析采用上述不同算法对内存利用率有什么不同的影响？如何解决因碎片而造成内存分配速度降低的问题？

根据运行结果，可以看到首次适应算法作业 8 分配失败，还有一些碎片没有利用起来，内存利用率较低；最佳适应算法成功全部分配，内存利用率高；最坏适应算法作业 7、8 分配都失败，还有大块碎片没有利用起来，内存利用率最低。尽可能的合并相邻的碎片，使用内存率较高的算法节省内存。

实验三　请求调页存储管理方式的模拟

一、实验目的

通过对页面、页表、地址转换和页面转换过程的模拟，加深对请 求调页系统的原理和实现过程的理解。

二、实验内容

1、编程模拟实现进程内存访问及页面置换算法

（1） 用 C 语言模拟一作业的执行过程，该作业共有 320 条指令。假设每个页面中可以存放 10 条指令，即该作业的逻辑地址空间为 32 页。分配给这个作业的内存块数为 4 个，开始时它的所有页都还没有调入内存。

（2） 模拟如下请求分页存储管理方式：

● 如果所访问的指令已在内存中，则显示其物理地址，并转下一条指令。

● 如果所访问的指令还未装入内存，则发生缺页，此时须记录缺页的次数，并将相应页调入内存；如果调页时所有内存均已装入，则需要按照置换算法进行页面置换。最后显示其物理地址，并转下一条 指令。

（3） 作业中指令的执行次序按如下原则随机生成：

● 50%的概率顺序执行下一条指令。

● 25%的概率跳转执行前面的指令。

● 25%的概率跳转执行后面的指令。

具体实现办法：

① 在[0, 319] 间随机选取一条指令开始启动执行，设其序号为 m；

② 顺序执行下一条指令，即序号为 m + 1 的指令；

③ 通过随机数跳转到[0, m]中的某条指令执行，设其序号为 m1；

④ 顺序执行下一条指令，即序号为 m1 + 1 的指令；

⑤ 通过随机数跳转到[m1 + 2, 319]中的某条指令处，设其序号为 m2；

⑥ 顺序执行下一条指令，即序号为 m2 + 1 的指令；

⑦重复执行②-⑥步，直到执行完 320 条指令为止。

（4） 编写程序分别实现 OPT、FIFO、LRU 置换算法。

2、使用上述模拟流程，分别统计上述三种置换算法缺页率，并 进行对比分析。

3、将分配给进程的物理块数改为 8 块，再分别统计上述三种置 换算法的缺页率，分析有什么变化规律。

三、实现思路

1.实验原理

请求分页系统是建立在基本分页的基础上的，为了能支持虚拟存储器功能而增加了请求调页功能和页面置换功能。相应地，每次调入和换出的基本单位都是长度固定的页面，这使得请求分页系统在实现上要比请求分段系统简单。请求分段系统在换进和换出时是可变长度的段，因此，请求分页便成为目前最常用的一种实现虚拟存储器的方式。

(1). 页表机制

在请求分页系统中所需要的主要数据结构是页表。其基本作用仍然是将用户空间中的逻辑地址变换为内存空间中的物理地址。由于只将应用程序的一部分调入内存，还有一部分仍在盘上，故需在页表中再增加若干项，供程序（数据）在换进、换出时参考。在请求分页系统中的每个页表项如下所示：

页号	物理块号	状态位 P	访问字段 A	修改位 M	外存地址

(2). 缺页中断

在请求分页系统中，每当所要访问的页面不在内存中时，便产生一次缺页中断，请求 OS 将所缺之页调入内存。缺页中断作为中断，同样需要经历诸如保护 CPU 现场、分析中断原因、转入缺页中断处理程序进行处理、恢复 CPU 现场等几个步骤。但缺页中断又是一种特殊的中断，它与一般的中断相比，有着明显的区别，主要表现在下面两个方面：在指令执行期间产生和处理中断信号。

通常，CPU 都是在一条指令执行完成后，才检查是否有中断请求到达。若有，便去响应，否则，继续执行下一条指令。然而，缺页中断是在指令执行期间，发现所要访问的指令或数据不在内存时所产生和处理的。一条指令在执行期间，可能产生多次缺页中断。所以，系统中的硬件机构应能保存多次中断时的状态，并保证最后能返回到中断前产生缺页中断的指令处继续执行。

(3). 地址转换

请求分页系统中的地址变换机构，是在分页系统地址变换机构的基础上，为实现虚拟存储器而增加了某些功能而形成的，如产生和处理缺页中断，以及从内存中换出一页的功能等等。 在进行地址变换时，首先去检索快表，试图从中找出所要访问的页。若找到，便修改页表项中的访问位。对于写指令，还需将修改位置成“1”，然后利用页表项中给出的物理块号和页内地址形成物理地址。地址变换过程到此结束。如果在快表中未找到该页的页表项时，应到内存中去查找页表，再根据找到的页表项中的状态位 P，了解该页是否已调入内存。若该页已调入内存，这时应将此页的页表项写入快表，当快表已满时，应先调出按某种算法所确定的页的页表项；然后再写入该页的页表项。若该页尚未调入内存，这时应产生缺页中断，请求 OS 从外存把该页调入内存。

2.设计方案

(1).先来先服务（FIFO）

置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。但是该算法会淘汰经常访问的页面，不适应进程实际运行的规律，目前已经很少使用。流程图如下。

具体实现过程为：定义变量 ptr。一开始先预调页填满内存。在这一部分，ptr 指向下一个要存放的位置。之后继续执行剩下的指令。此时，ptr 表示队列最前面的位置，即最先进来的位置，也就是下一个要被替换的位置。ptr 用循环加，即模拟循环队列。

(2). 最近最少使用法（LRU）

置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。 流程图如下。

LRU 算法普偏地适用于各种类型的程序，但是系统要时时刻刻对各页的访问历史情况加以记录和更新，开销太大，因此 LRU 算法必须要有硬件的支持。

具体实现如下：定义数组 ltu[],即 last_time_use 来记录该页最近被使用的时间。定义变量 ti 模拟时间的变化，每执行一次加一。本次实现未预调页，而是直接执行所有指令。若当前需要的页没在内存里，就寻找最近最少使用的页，

也就是 ltu[]最小的页，即最近一次使用时间离现在最久的页，然后替换掉它。或者在内存还未满时，直接写入，这个以初始化内存里所有页为-1 来实现。若已经在内存里了，则只遍历内存内的页，把当前页的最近使用时间改一下即可。

(3). 最佳页面替换法（OPT）

置换以后不再被访问，或者在将来最迟才回被访问的页面，缺页中断率最低。但是该算法需要依据以后各业的使用情况，而当一个进程还未运行完成是，很难估计哪一个页面是以后不再使用或在最长时间以后才会用到的页面。所以该算法在实际情况中几乎是不能实现的。但该算法仍然有意义，作为很亮其他算法优劣的一个标准。流程图如下。

具体实现如下：定义数组 ntu[]，即 next_time_use 来记录下一次被使用的时间，即将来最快使用时间。初始化为-1。开始时预调页填满内存里的页。同样利用变量 ptr 来表示下一个要存放的位置从而控制预调页的过程。接着初始化 ntu 数组为-1。然后求出每一页下一次被使用的指令号，以此代替使用时间。如果所有剩下的序列都没有用该页时，则还是-1.这种值为-1 的页显然是最佳替换对象。然后执行所有剩下的指令。当该页不在内存里时，遍历 ntu 数组，遇到-1 的直接使用该页，没有则用 ntu[]值最大的，也就是最晚使用的。无论该页在不在内存里，因为这一次已经被使用了，所以都应该更新这个页的 ntu[]，只需往前看要执行的页流，记录下第一个遇到的该页即可。如果没有找到同样添-1 即可。

四、主要的数据结构

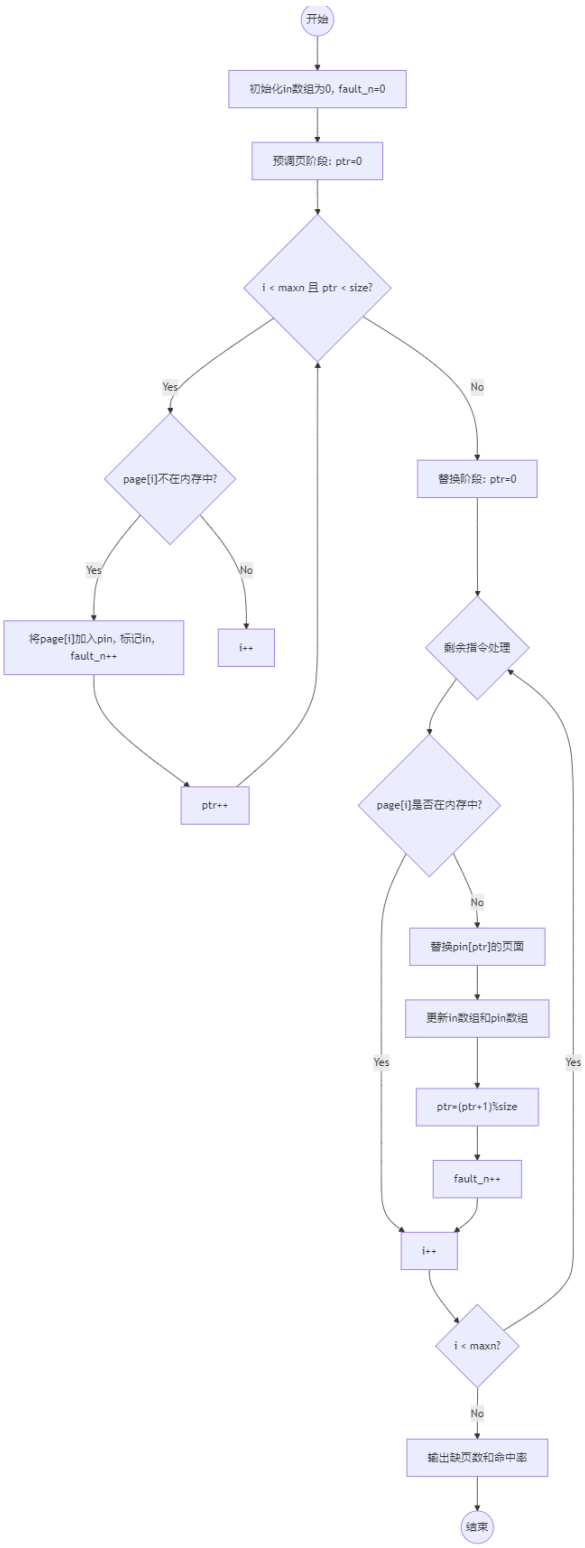
```
#define maxn 320
#define max (maxn + 20)
#define maxp (max / 10)

int inst[max];
int page[max];
int size;

int in[maxp];
```

int pin[maxp];

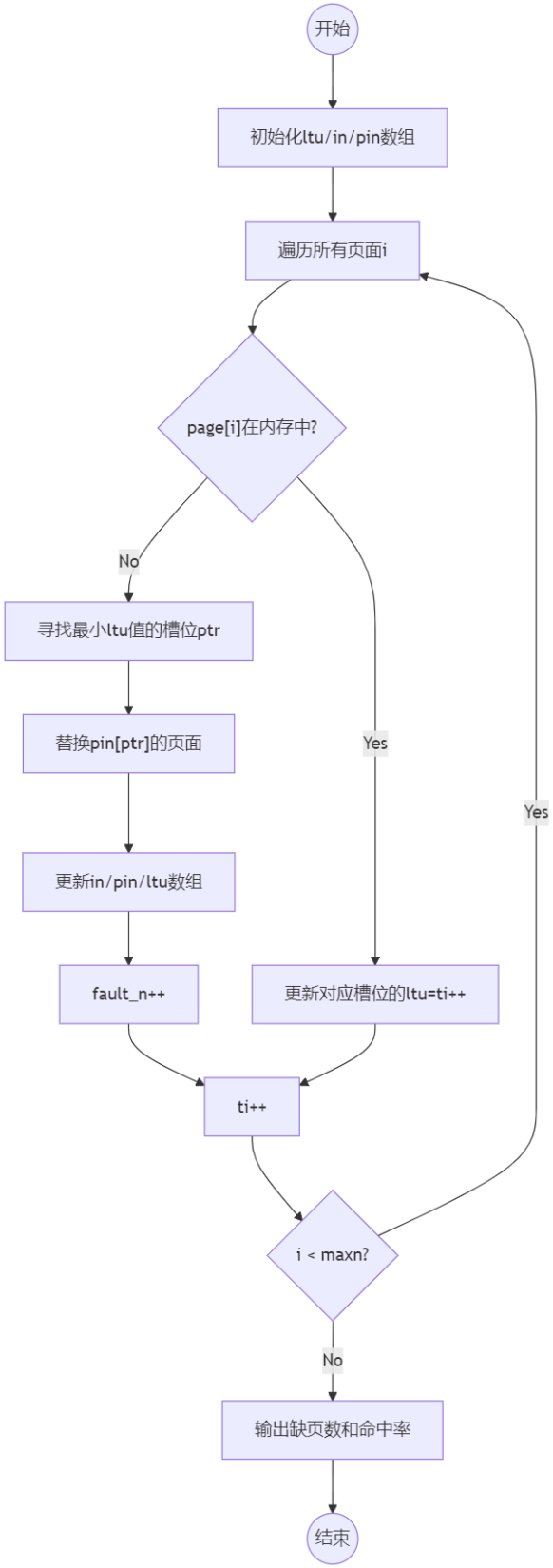
各函数调用关系：

五、算法流程图

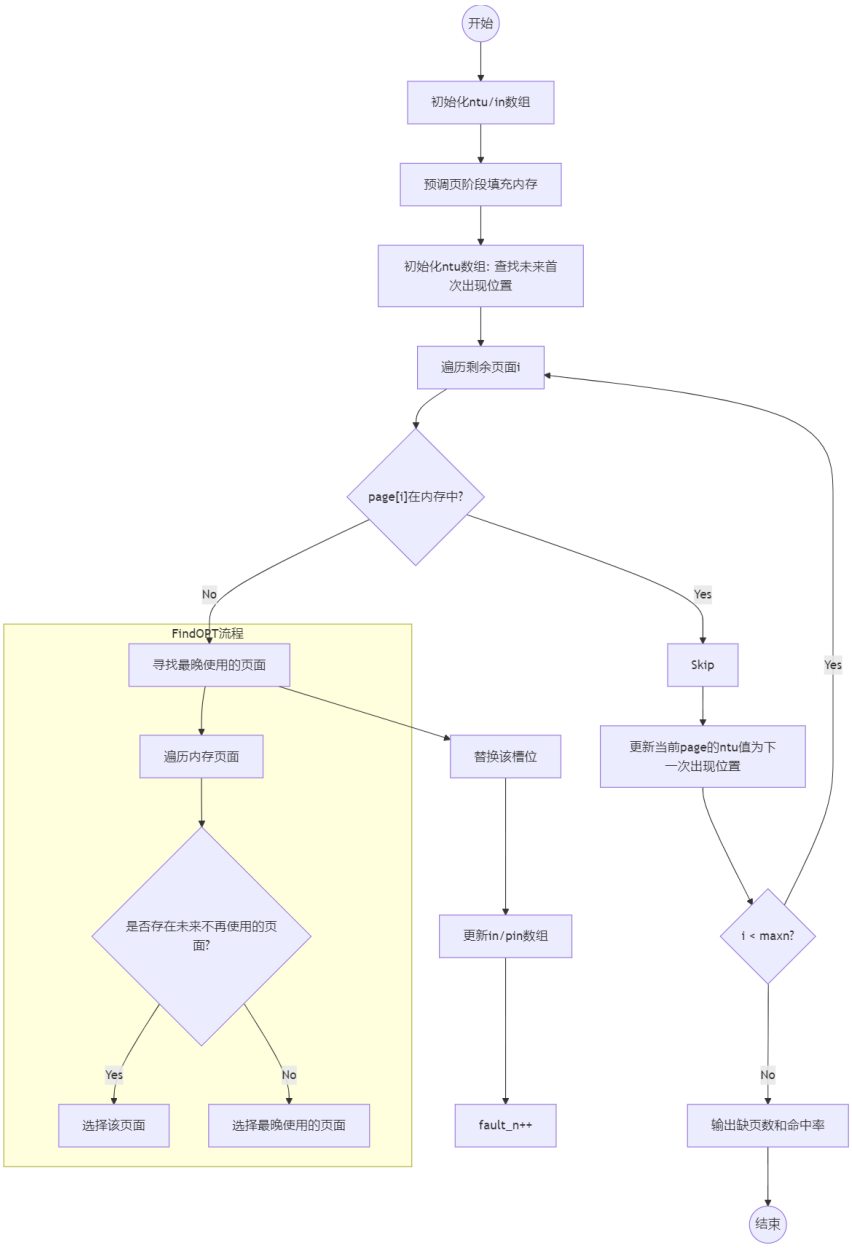
FIFO_solve() 函数流程图：



LRU_solve() 函数流程图：



OPT_solve() 函数流程图：


开始
初始化ntu/in数组
预调页阶段填充内存
初始化ntu数组：查找未来首次出现位置
遍历剩余页面i
page[i]在内存中?
No
Yes
FindOPT流程
寻找最晚使用的页面
遍历内存页面
是否存在未来不再使用的页面?
Yes
No
选择该页面
选择最晚使用的页面
替换该槽位
更新in/pin数组
fault_n++
Skip
更新当前page的ntu值为下一次出现位置
i < maxn?
Yes
No
输出缺页数和命中率
结束

六、运行与测试

实验总得分：**25.00**

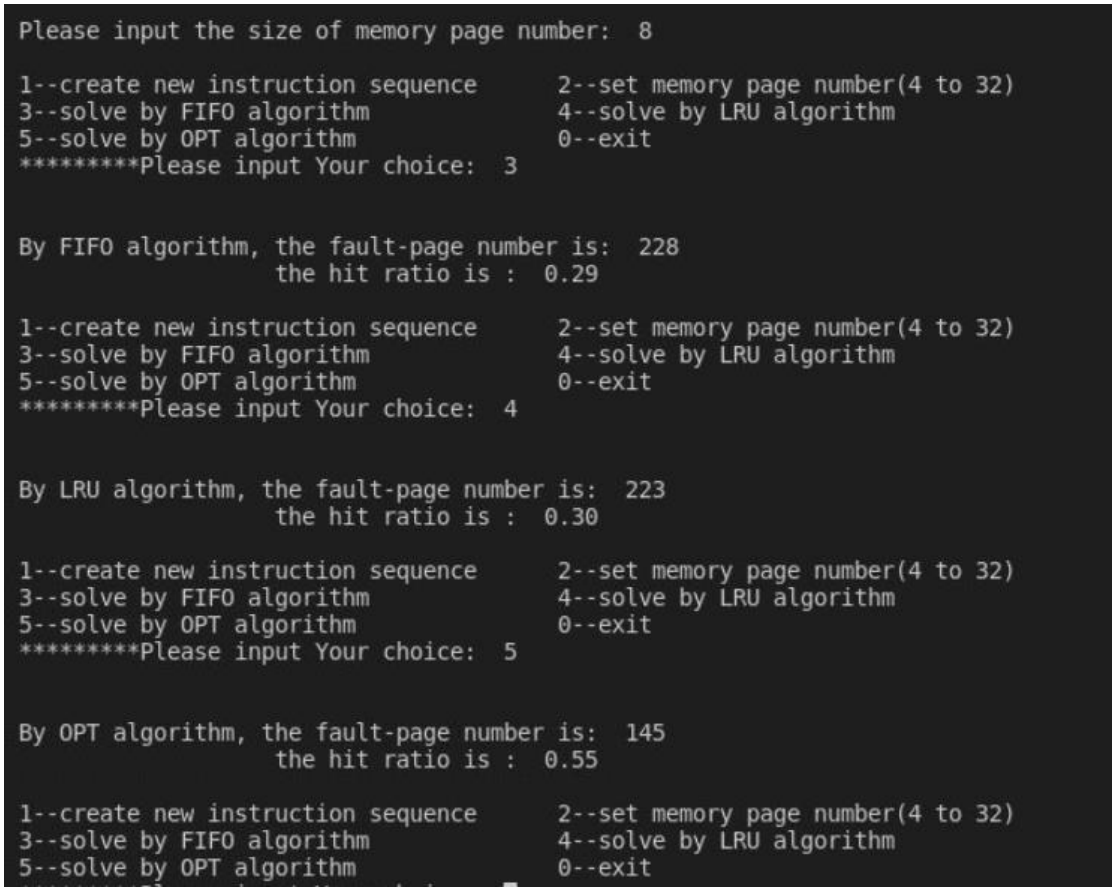
Test Pass | 步骤得分：**100** | 2025-05-27 15:55:59

当前步骤自动评测得分（百分制）： 100

> — 当前分数是百分制，最后转换为实际得分。

Congratulations on getting 100 points!!!

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

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:  264
                   the hit ratio is :  0.18

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:  270
                  the hit ratio is :  0.16

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:  203
                  the hit ratio is :  0.37

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
```



1.使用上述模拟流程，分别统计上述三种置换算法缺页率，并进行对比分析。

答： 可以看到击中率 OPT > LRU > FIFO，这也代表了它们的性能排序，缺页率 FIFO > LRU > OPT。

2.将分配给进程的物理块数改为 8 块，再分别统计上述三种置换算法的缺页率，分析有什么变化规律。

答：可以看到缺页率 OPT 为 0.44，LRU 为 0.70，FIFO 为 0.71，可以看到对比起 4 页记忆页，缺页率下降了。

七、改进的方向

其实可以增加测试的例子，这样会显得更直接，或者直接通过画图和列表等实验的形式来体现出 OPT，LRU 和 FIFO 的对比和缺页率。同时还应该给一个例子添加详细动画或者代码 debug 演示，这样能更直观地看出交换页内项目的过程，更方便理解。

思考 （1） 如果增加分配给作业的内存块数，将会对作业运行过 中的缺页率产生什么影响？

答：增加内存块数会降低缺页率，但降低效果越来越低。

（2） OPT、FIFO、LRU 哪种置换算法的性能好？为什么一般情 况下 LRU 具有比 FIFO 更好的性能？

答：从结果来看，OPT > LRU > FIFO。因为 LRU 通过判断记录使用次数，优先

保留活跃页面，而 FIFO 可能会置换出刚进入但频繁使用的页面。

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（含目录）。

read：读文件数据。

write：写入文件数据。

delete：删除文件。

rm：删除目录（支持空目录和非空目录）。

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. 实验原理

文件是由操作系统定义并实现的一种抽象数据类型，它是逻辑记录的序列。一个记录可能是一行数据或一个字节。操作系统可支持不同的记录类型，当然它们应

由相应的应用程序来定义和解释。

文件系统最基本的目标是实现文件按名存取，这主要是通过文件系统的目录管理功能来实现。文件系统所追求的的最重要的目标是提高对文件的存取速度。一个较为完整的文件系统应具有以下功能。

①文件读写管理：提供合适的文件存取方法，根据用户请求尽快地完成文件的读写管理。

②文件目录管理：建立和维护目录，实现对文件按名存取，提供快速的目录检索。

③文件存储空间管理：为每个文件分配必要的外存空间，提高外存利用率，提高文件系统的工作速度。

④文件保护与共享：实现多个用户对同一个文件的共享访问，并提供存取控制等保护措施。

⑤提供方便的接口：通过系统的命令接口和程序接口，提供一组可供用户使用的文件系统，方便用户对文件的使用和管理。

⑥文件系统的可靠性与一致性：提供文件的备份、转储与恢复机制，并能验证文件的正确性，具有一定的差错恢复能力。

(1).文件

操作系统管理的文件是指外部存储器中具有符号名的一组相关信息的集合。通常文件在逻辑上由若干个记录组成。记录是一些相关数据项的集合，而数据项是数据组织中可以命名的最小逻辑单位，用于描述一个实体的某种属性。因此，文件也是具有符号名的记录的集合，特别是在早起的系统和现在的数据库管理系统中。

文件包括文件体和文件说明两部分。

①文件体：指文件本身的信息。这是操作系统管理的一种软资源对象。

②文件说明：指文件存储和管理信息，例如，文件名、文件内部标识、文件存储地址、访问权限和访问时间等。这是操作系统为管理文件对象而需要的管理控制信息，也可看成是文件的属性，通常存放在文件目录之类的数据结构中。

文件的表示范围很广，系统或用户可以将具有一定功能的程序或数据集合命名后存储为一个文件。例如，一个命名的源程序、目标程序、一批数据，以及系

统程序都可看作文件。在许多操作系统中，设备也被看作赋予特殊文件名的文件。这样，系统可以对设备和文件实施统一管理，既简化了系统设计又方便了用户。

(2).文件系统

操作系统中负责存取和管理文件信息的模块称为文件系统。它用统一的方式管理信息的存储、检索、更新、共享和保护，并为用户提供一整套方便有效的文件使用和操作方法。文件这一术语不但反映了用户概念中的逻辑结构，而且与存放它的辅助存储器的存储结构也是紧密相关的。

从系统角度看，文件系统是对文件的存储空间进行组织、分配，负责文件的存储并对存入的文件进行保护。检索的系统。它负责为用户建立。撤销、读写、修改和复制文件。从用户角度看，文件系统主要实现了按名存取。也就是说，当用户要求系统保存一个已命名文件时，操作系统以一定的格式将用户的文件存放到文件存储器中适当的位置；当用户要使用文件时，系统根据用户所给的文件名能够从文件存储器中找到所要的文件。对用户来说它提供了一种便捷的存取信息的方法：按文件名存取信息，无须了解文件存储的物理位置。从这种意义上讲，文件系统是用户与外存的接口，这里把设备管理模块看作文件系统的底层。更严格的说法是文件系统位于用户和设备驱动程序之间。一种文件系统的样例结构如图所示。

(3).目录

文件目录主要用于文件的描述与控制，实现稳健的按名存取和文件信息共享和保护。文件目录项，又称文件控制块（File Control Block，FCB）随文件的建立而建立，随文件的删除而消亡，它对于文件，就像 PCB 对进程一样。

2.设计方案

使用文件的目的是存储信息并方便以后检索。不同的系统提供不同的文件操作命令和系统调用，其中有很多命令早已为广大的计算机用户所熟悉。

从某种意义上说，用户使用计算机的主要目的就是要访问存储在外存中的文件。而要访问这些文件，可以通过文件系统提供的两类接口进行。第一类是与文件有关的一组操作命令或作业控制语言中与文件有关的一组 JCL 语句，例如，UNIX 系统中的 cat、cp、find、mv、rm 等命令，MS-DOS 中的 type、copy、delete、rename、attrib 等命令，Windows 中的搜索工具集成了建立、删除、复制、粘贴、

改名等功能于一身的资源管理器或我的电脑命令工具等，这些构成了必不可少的命令级的文件系统人机接口。第二类是提供给用户程序使用的一组文件系统调用，它们构成了程序级的用户和文件系统的接口。通过这些接口，用户能够获得文件系统的各种服务，从而很轻松地完成所需要的各种文件操作，例如创建、打开、读、写、关闭和删除文件等。实际上，正是文件系统提供的这两类接口，使用户对文件的操作更加简单、方便和有效。现在请你在提供的框架下，实现操作系统提供的最通常的一些文件类系统调用。

①CREATE，创建文件。创建无任何数据的文件。

②DELETE，删除文件。当不再需要某个文件时，必须删除它以释放外存空间。

③READ，读文件。在文件中读数据。一般，读出的数据来自文件读写指针的当前位置。在每次读写完成后，读写指针会随着信息读写的数据量自动后移相应数值。

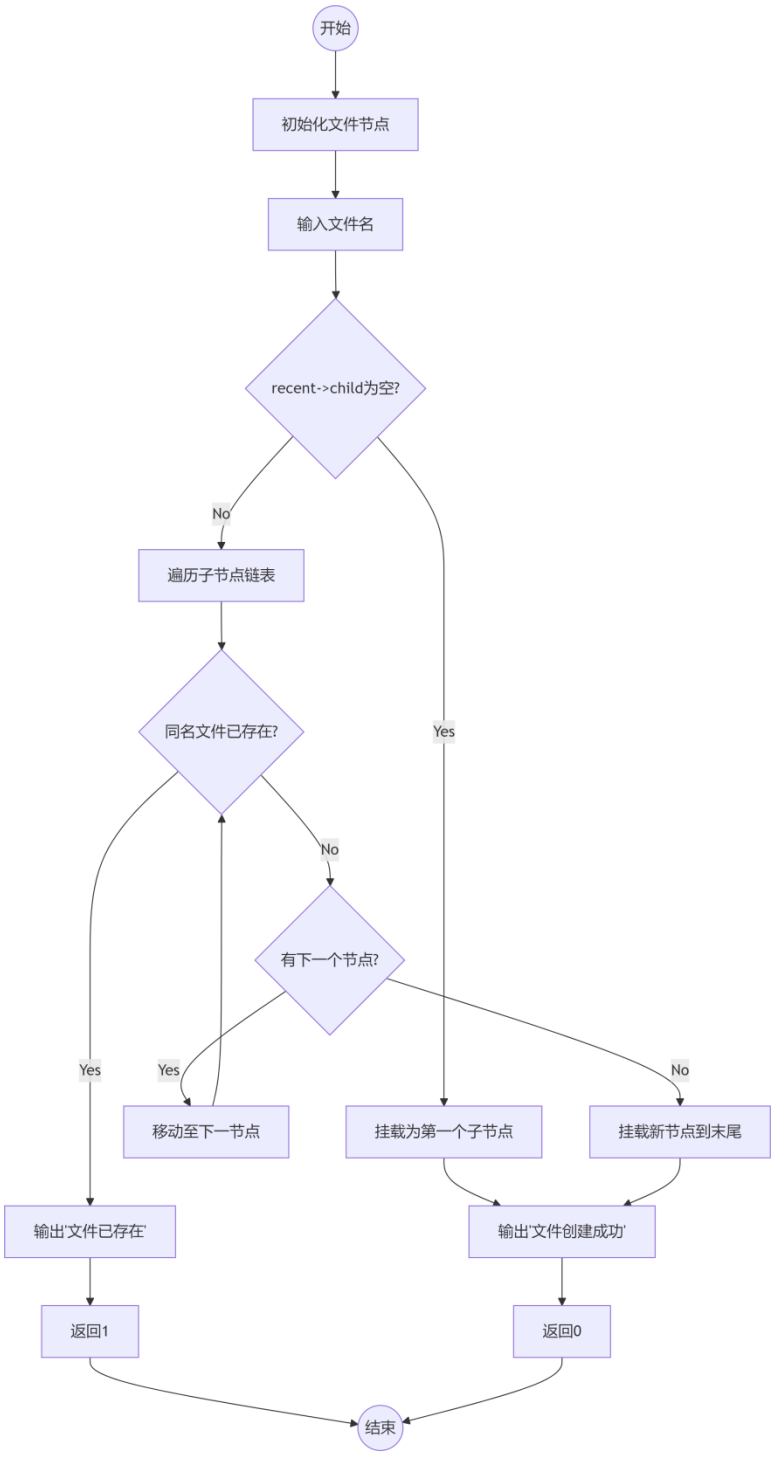
④WRITE，写文件。在文件中写数据。写操作一般也是从文件读写指针的当前位置开始。如果当前位置是文件末尾，则文件长度增加；如果当前位置在文件中间，则现有数据被覆盖，并且永远丢失掉。

四、主要的数据结构

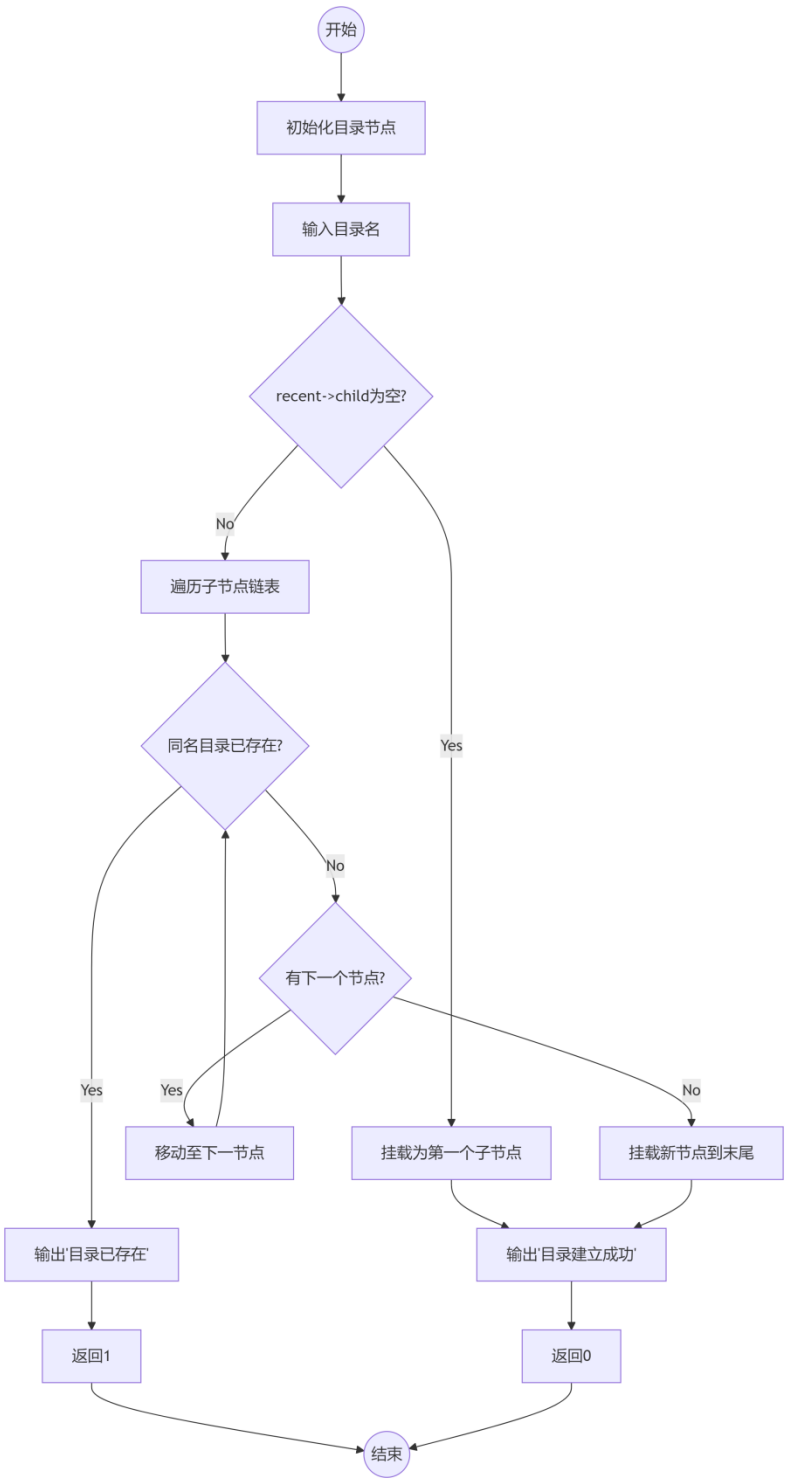
```
struct filenode
{
    char filename[FILENAME_LENGTH];
    int isdir;
    char content[255];
    struct filenode *parent;
    struct filenode *child;
    struct filenode *prev;
    struct filenode *next;
};
```

五、算法流程图

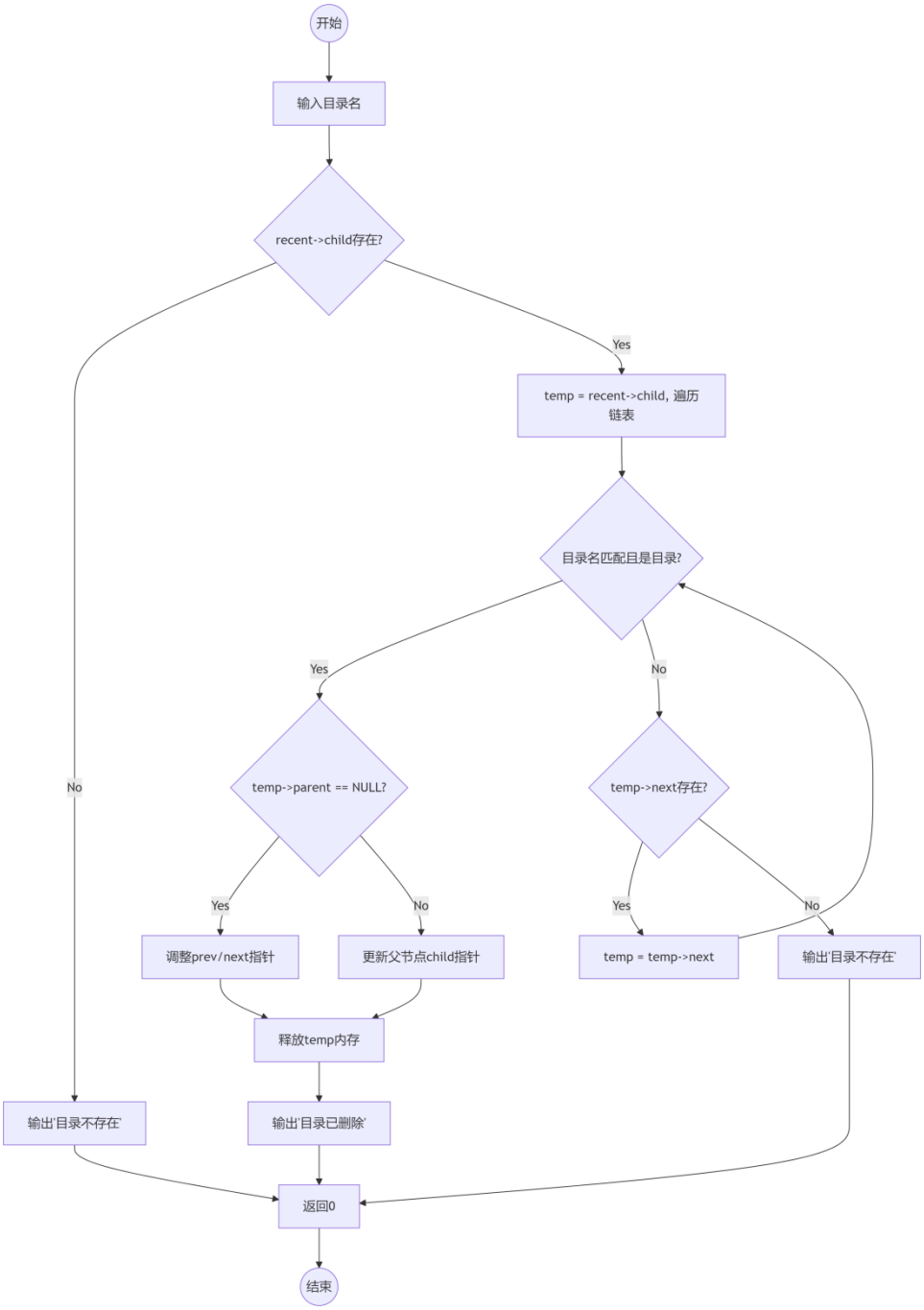
create()函数流程图：


开始
初始化文件节点
输入文件名
recent->child为空?
No
Yes
遍历子节点链表
同名文件已存在?
No
有下一个节点?
Yes
Yes
No
移动至下一节点
挂载为第一个子节点
挂载新节点到末尾
输出'文件已存在'
输出'文件创建成功'
返回1
返回0
结束

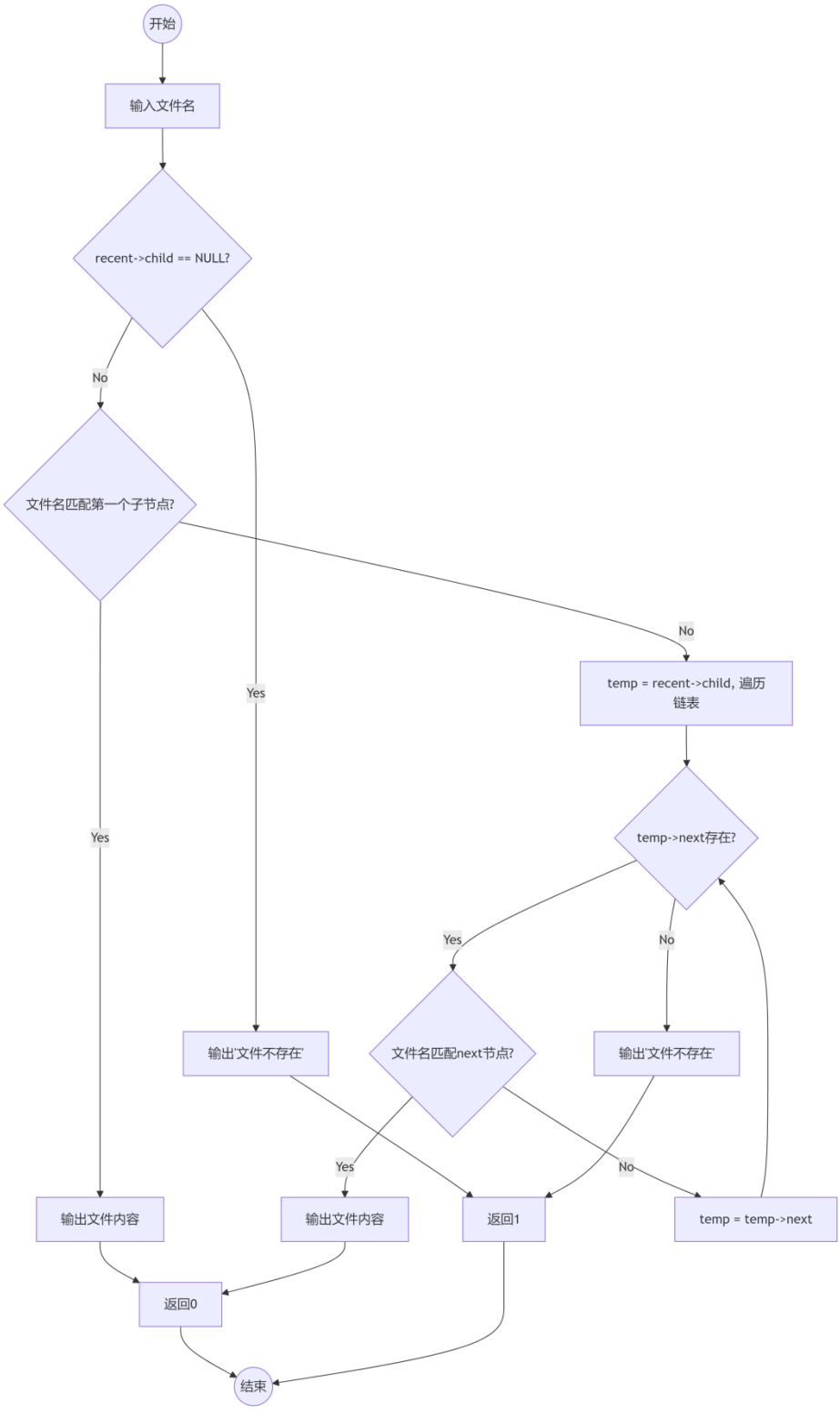
mkdir()函数流程图：


开始
初始化目录节点
输入目录名
recent->child为空?
No
Yes
遍历子节点链表
同名目录已存在?
No
有下一个节点?
Yes
Yes
No
移动至下一节点
挂载为第一个子节点
挂载新节点到末尾
输出'目录已存在'
输出'目录建立成功'
返回1
返回0
结束

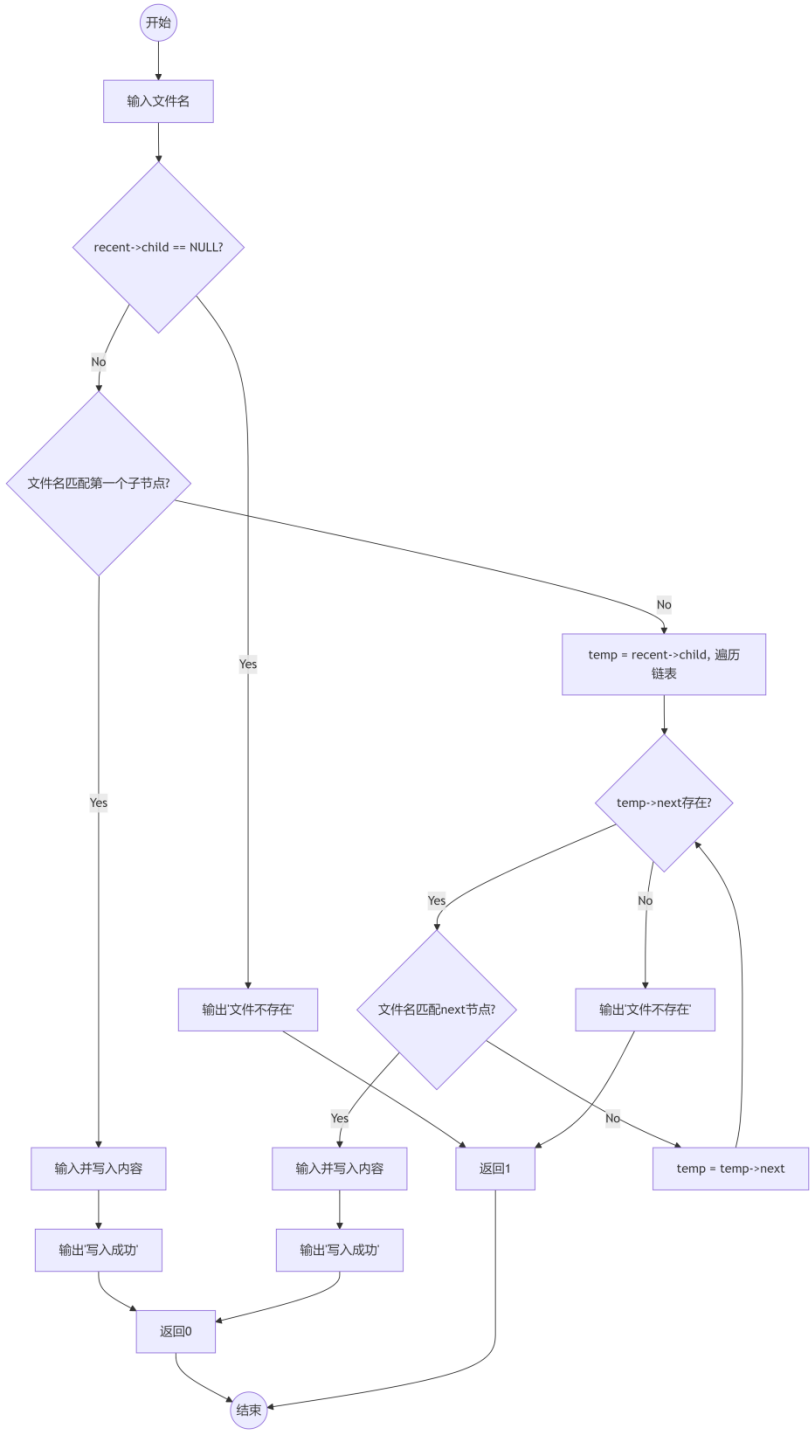
rm()函数流程图：


开始
输入目录名
recent->child存在?
No
Yes
temp = recent->child, 遍历链表
目录名匹配且是目录?
Yes
No
temp->parent == NULL?
temp->next存在?
Yes
No
Yes
No
调整prev/next指针
更新父节点child指针
temp = temp->next
输出'目录不存在'
释放temp内存
输出'目录不存在'
输出'目录已删除'
返回0
结束

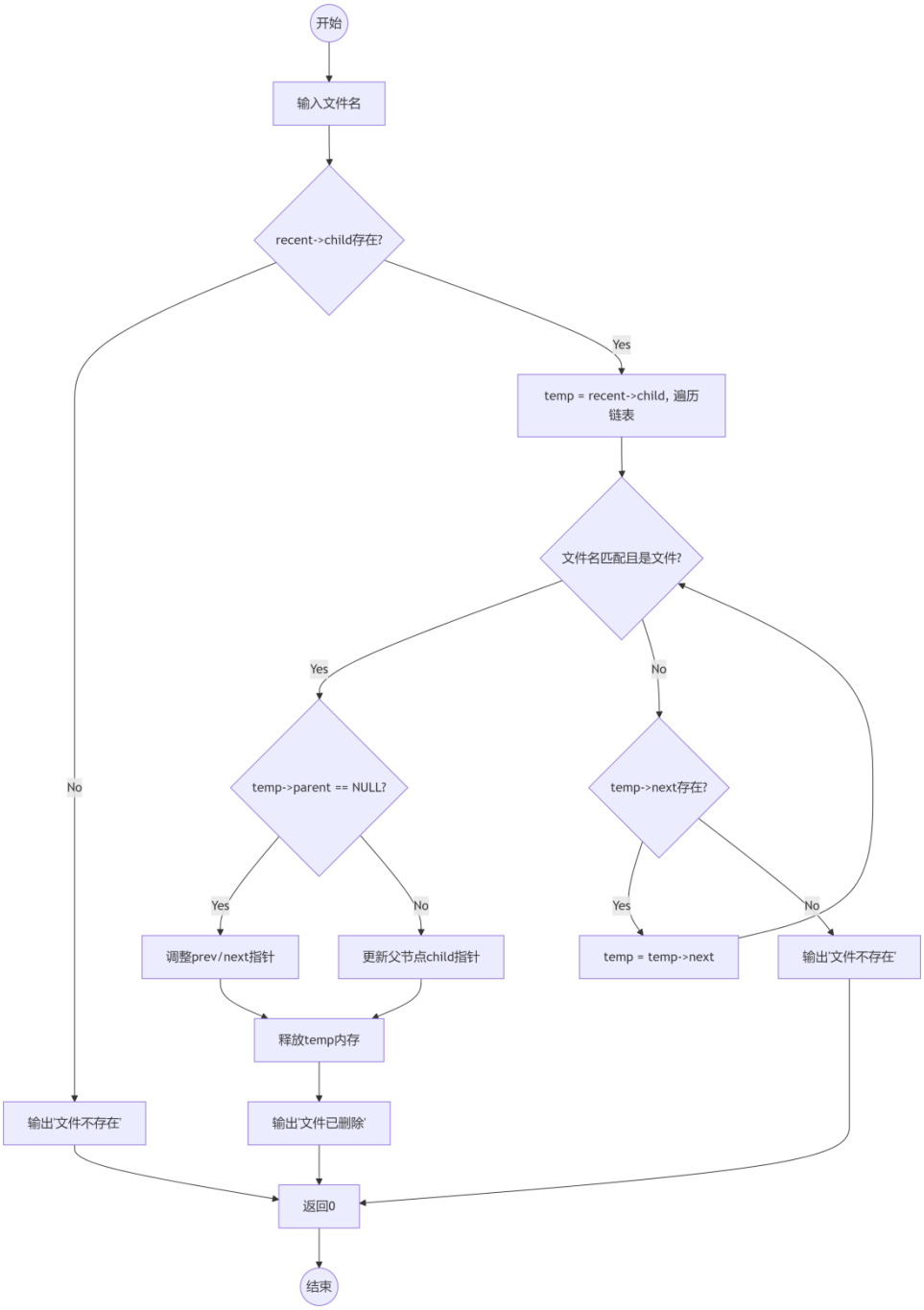
read()函数流程图：


开始
输入文件名
recent->child == NULL?
No
Yes
文件名匹配第一个子节点?
No
Yes
temp = recent->child, 遍历链表
temp->next存在?
Yes
No
输出'文件不存在'
文件名匹配next节点?
输出'文件不存在'
Yes
No
输出文件内容
输出文件内容
返回1
temp = temp->next
返回0
结束

write()函数流程图：


开始
输入文件名
recent->child == NULL?
No
Yes
文件名匹配第一个子节点?
No
Yes
temp = recent->child, 遍历链表
temp->next存在?
Yes
No
输出'文件不存在'
文件名匹配next节点?
输出'文件不存在'
Yes
No
输入并写入内容
输入并写入内容
返回1
temp = temp->next
输出'写入成功'
输出'写入成功'
返回0
结束

del 函数流程图：


开始
输入文件名
recent->child存在?
No
Yes
temp = recent->child, 遍历链表
文件名匹配且是文件?
Yes
No
temp->parent == NULL?
temp->next存在?
Yes
No
Yes
No
调整prev/next指针
更新父节点child指针
temp = temp->next
输出'文件不存在'
释放temp内存
输出'文件不存在'
输出'文件已删除'
返回0
结束

六、运行与测试

提交评测

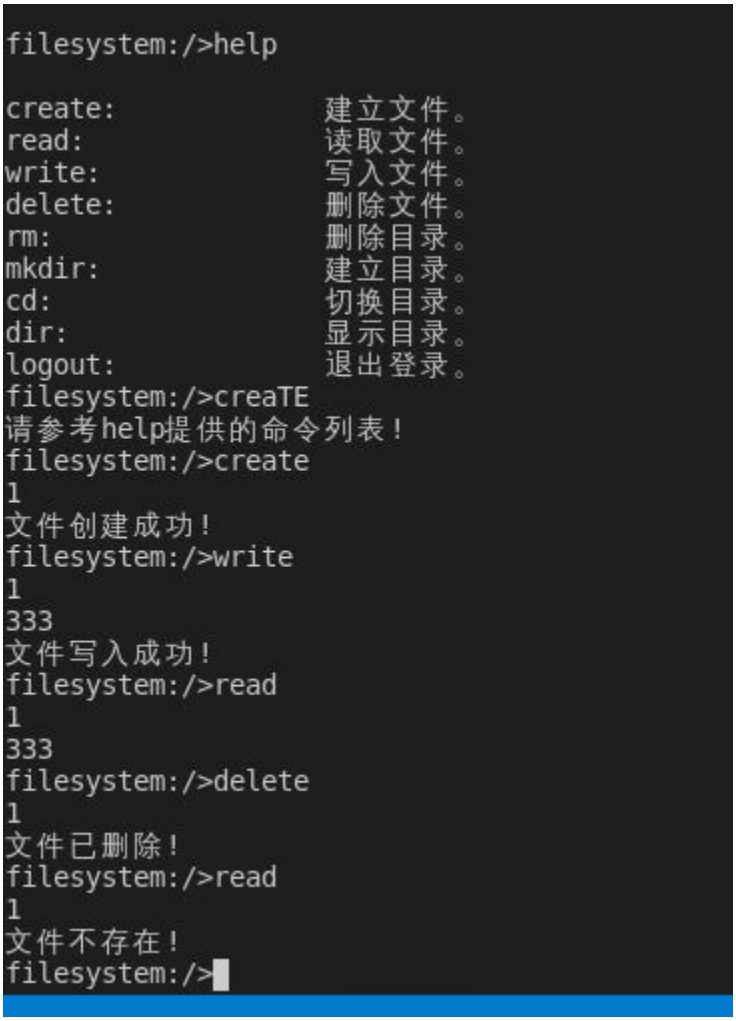
实验总得分：**25.00**

Test Pass | 步骤得分：**100** | 2025-05-27 15:55:06

当前步骤自动评测得分（百分制）： 100

> — 当前分数是百分制，最后转换为实际得分。

Congratulations on getting 100 points!!!



1. 可选实现如下相关操作：

- open：打开文件。
- close：关闭文件。

代码：

```
int open_file(const char *filename) {
    for(int i=0; i<MAX_OPEN_FILES; i++) {
        if(!open_files[i]) {
            FileNode *file = find_node(filename);
            if(file) {
                open_files[i] = file;
                file->ref_count++;
                return i;
            }
        }
    }
    return -1;
}

void close_file(int fd) {
    if(fd >=0 && fd < MAX_OPEN_FILES) {
        open_files[fd]->ref_count--;
        open_files[fd] = NULL;
    }
}
```

2.可选实现文件系统的持久化。

```
void save_filesystem(const char *filename) {
    FILE *fp = fopen(filename, "wb");
    if(!fp) return;

    FileNode *current = root;
```

```
    while(current) {
        fwrite(current, sizeof(FileNode), 1, fp);
        current = current->next;
    }
    fclose(fp);
}
```

七、改进的方向

可以增加更多的功能，比如显示子目录，再比如新建 text 文件并进行修改，这样能更好的对文件的操作进行理解。同时还可以使用更高效的存储方式，我个人认为更高效的数据结构有哈希表，字典，try，这些都可以增加用户访问文件的效率。

思考

（1） 如果引入磁盘索引结点，上述实现过程需要作哪些修改？

答：则需要修改链表结构，读写操作需通过磁盘索引结点；需要增加索引结点管理模块，引入索引结点位图管理索引结点分配；还需将原融合的 FCB 拆分为目录项和索引结点等等。

（2） 如果设计的是一个多用户文件系统，则又要进行哪些扩充？

答：需要增加权限管理，实现用户登录模块，用户上传文件，管理不同的文件，不同的用户有不同的文件库并且可以自己进行管理等等。