

## 实验 04、迷你区块链系统

王艺添 2021202434

### 1. 背景

在实验 02 中，我们从文件中读取数据，构造了由一组区块构成的链表。本实验要求在此基础上，实现一个迷你区块链系统。包括多个节点之间通过“随机”共识来维护一条一致的链，完成消息“收发”处理，以及查询功能。

因此需要用 C 语言写多个程序，同时运行来共同维护区块链。

### 2. 目的

本实验的主要目的在于：

- 练习队列操作和文件操作
- 为后续实验做准备

### 3. 问题描述

搭建一个由三个“节点”（程序）构成的微型区块链系统。

节点 1：模拟一个终端客户，每隔一定时间（比如 100 毫秒）随机向另外两个程序发送交易请求或查询请求。交易数据从实验 02 的数据集中生成。每条请求将“插入”到对应节点的“客户消息队列”尾部。

节点 2 和节点 3：模拟两个区块链节点。这些交易所引用的脚本，并输出脚本运行结果。

### 4. 功能描述

首先展示一下文件夹，虽然文件夹内容很多，但是每一个都有用，下面将一一解释。

- |             |                             |
|-------------|-----------------------------|
| ✓ backup    | ✓ Node2.c                   |
| ✓ KMP       | ✓ Node2.exe                 |
| ✓ message1  | ✓ renew.c                   |
| ✓ message2  | ✓ renew.exe                 |
| ✓ Agent.c   | ✓ transactions.csv          |
| ✓ Agent.exe | ✓ transactions_generate.csv |
| ✓ cJSON.c   |                             |
| ✓ cJSON.h   |                             |
| ✓ Node1.c   |                             |
| ✓ Node1.exe |                             |


首先是四个.exe文件，Agent.exe模仿用户节点，向另外两个程序发送交易请求或查询请求。Node1.exe和Node2.exe除了访问与储存地址不同以外逻辑一模一样，用来模仿区块链节点，用来处理终端客户发来的信息。由于每次运行结束后（可能因为终端循环的存在没法运行结束），csv的信息无法复原，renew.exe用于更新数据，即将backup中的数据更新到其他文件夹，可以使程序重新正常运行，并且赋予了两种更新模式，demo更新和2009更新，用于小样本测验和大样本测验。


接着是.c和.h文件，由于生成区块消息需要将结构体（对象）发送到另一个程序，因此每一条消息都用.json实现，这需要使用cJSON包实现结构体和.json文件的相互转换。于是从<https://github.com/DaveGamble/cJSON>上下载cJSON，并将cJSON.c和cJSON.h丢到了程序所需的文件夹里，并在需要的程序里面#include“cJSON.h”。（这样会导致vscode无法直接对程序进行编译，在实验第五部分我会解释是怎么解决的）

最后是各个.csv文件，csv文件用来实现队列，每次入队则在文档的最后插入一行信息，出队则删除第一行信息（由于csv有表头，应该是删除第二行信息），并将下面的信息依次往上移动一行，相当于用了一个没有范围限制不会上溢的顺序队列（可以这么理解）。

transactions.csv用来描述交易池，Node1.exe和Node2.exe均可根据用户指令对交易池进行出队和入队操作，transactions\_generate.csv用于产生随机生成的交易，由于每次renew后transactions\_generate.csv里面的顺序都会被随机打乱，保证了Agent.exe交易生产的随机性。

message1和message2用于存储Node1和Node2获取的消息，并且可以被三个.exe文件读取或改写，其中agent\_message.csv用于保存Agent.exe发送的消息，block\_message.csv用于保存另一个节点发送的区块消息。

 agent\_message1.csv

 block\_message1.csv

由于从Agent随机生成交易在发送信息和Node接受信息后再随机生成交易等效，且后者会避免区块交易的使用，agent\_message.csv只用于存储sign（交易或查询）。Node2生成区块信息后发送给Node1，包含两个部分，首先需要把区块包装成.json发送到message1文件夹，再将其地址发送到block\_message1.csv中，做入队操作，这样block\_message1.csv相当于一个存满了区块指针的队列，多线程的交互功能基本得以实现。

我没想到这个区块链和 KMP 有什么关系，所以单独写了个算法丢在 KMP 文件夹里。

\*\*\*不要改变各个文件的位置，不然程序很有可能会崩溃。

接下来是三个节点和 **renew** 的部分实现代码逻辑。

renew.c 包含两个主要的函数：

```
void clearDirectory(const char *path);  
void copyFile(const char *sourcePath, const char *destinationPath);
```

第一个函数用于清除指定文件夹里的所有内容；

第二个函数用于将 backup 中的文件拷贝到目的文件夹。

Agent.c 包含两个主要函数：

```
void loopActivate();  
void writeToCSV(char* filename, char* operation);
```

第一个函数用于循环发送随机信息（可选择手动或自动）；

第二个函数用于将信息写入 csv 文件，被第一个函数调用。

Node.c 相对来说非常复杂，包含下面几组函数：

```
// csv 文件读写函数  
void deleteLines(const char *filename, int n);  
void writeFilenameToCSV(const char *filename);  
char* getMessage(const char* filename);  
  
// json 文件读写函数  
cJSON* transactionToJSON(Transaction* transaction);  
cJSON* blockToJSON(Block* block);  
void writeJsonToFile(cJSON *jsonObject, const char *filename);  
void parseTransactionFromJSON(cJSON* jsonTransaction, struct  
Transaction* transaction);  
Block* readBlockFromJSONFile(const char* filename);  
  
// 哈希值生成函数  
unsigned long hash_djb2(unsigned int num);  
  
// 区块链函数  
Node_Transaction* select_Transaction(int number_select);  
Block* New_Block(Block* list, Node_Transaction* select);  
Block* Add_Block(Block* list);  
Block* createBlocklist();  
void lengthLinkedList(Block* list);  
int countTransaction(Block* list);  
void infoLinkedList(Block* list);
```

```
// 功能实现函数
void loopActivate(Block* list);
```

csv 文件读写函数：用于删除 csv 前 n 行（出队 n 次）、将文件名字写入 csv 文件（区块指针入队）、获得 csv 中第一行文本内容（消息出队）；

json 文件读写函数：用于将单条交易信息写入 json、将单个区块信息写入 json、将 json 对象写成 json 文件、读取 json 中文件中的 transaction 信息、读取 json 文件中的区块信息并返回 Block\*（一个指向区块的指针）；

哈希值生成函数：用 djb2 方法生成 hash 值；

区块链操作函数：用于将选择的 transaction 组织成链表（方便临时存储多条 transaction 信息）、给区块链插入一个区块（尾插）、根据随机的 transactions 生成一个区块、初始化一个带头结点的循环双链表用于储存区块信息、计算区块链区块长度、计算区块链包含交易数、展示区块链信息；

功能实现函数：循环实现中奖和不中奖，中奖插入区块并发送信息，不中奖处理消息队列。

为了展示方便，在实际运行代码时我把原来的自动随机消息改成了手动发送指定消息，有利于测试功能是否完全实现。

下面是程序运行的展示。

首先同时点开所有.exe 文件（不包含 renew.exe）

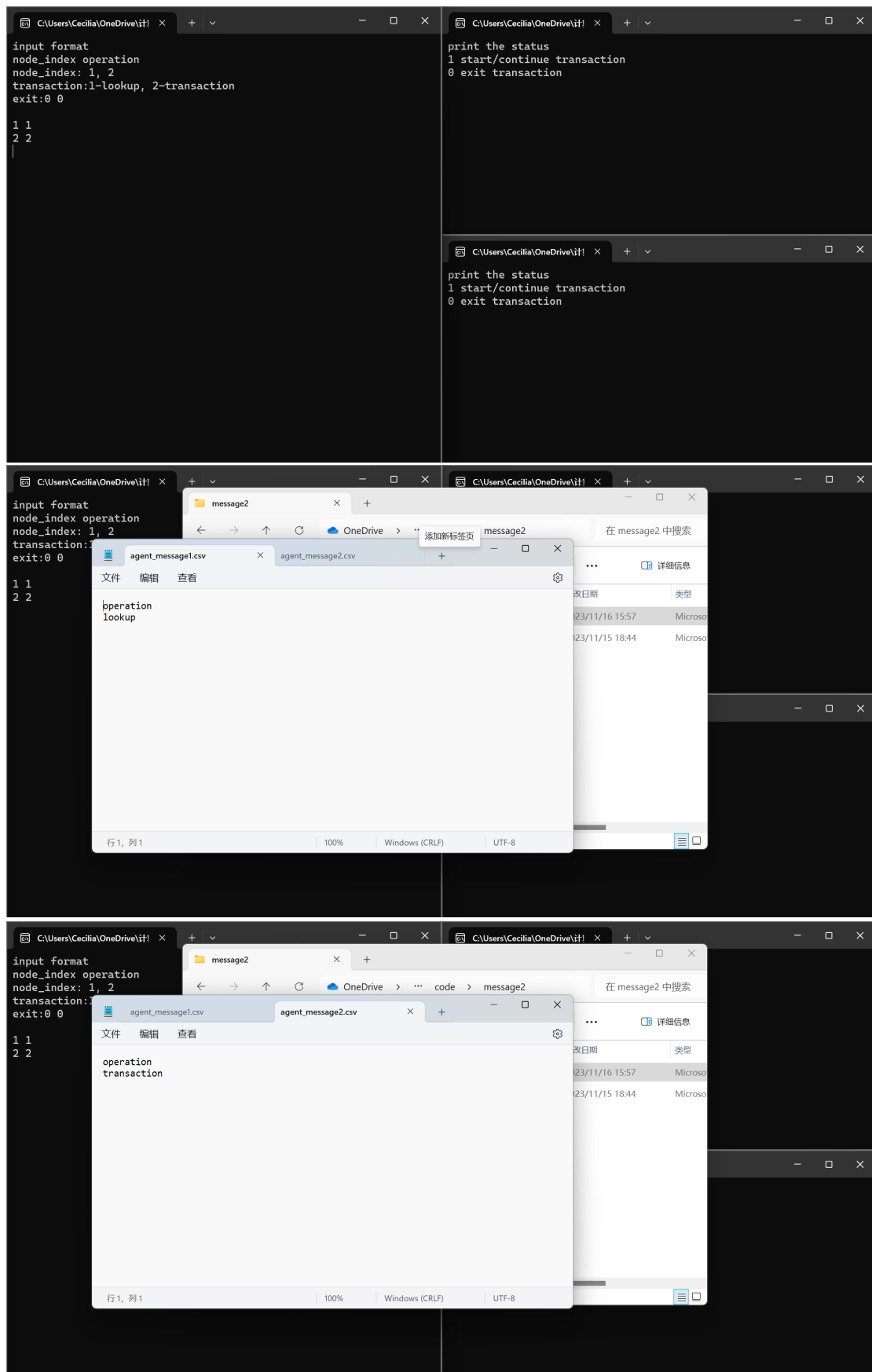
```
input format
node_index operation
node_index: 1, 2
transaction:1-lookup, 2-transaction
exit:0 0

print the status
1 start/continue transaction
0 exit transaction

print the status
1 start/continue transaction
0 exit transaction
```

左边是 Agent 节点，右边两个是 Node 节点（上面 2 下面 1），首先测试 Agent 节点是否能正常发送信息，我们给 Node1 发送一个查询请求，给 Node2 发送一个交易请求。

输入方法均有提示，我们再来检查文件夹：



我们看到 agent\_message 已经更新内容，说明 Agent.exe 成功将用户消息发送至两个节点。下一步我们将 Node1 运行一次：

```
C:\Users\Cecilia\OneDrive\计 x + - □ x
input format
node_index operation
node_index: 1, 2
transaction:1-lookup, 2-transaction
exit:0 0

1 1
2 2

C:\Users\Cecilia\OneDrive\计 x + - □ x
print the status
1 start/continue transaction
0 exit transaction

C:\Users\Cecilia\OneDrive\计 x + - □ x
print the status
1 start/continue transaction
0 exit transaction

1

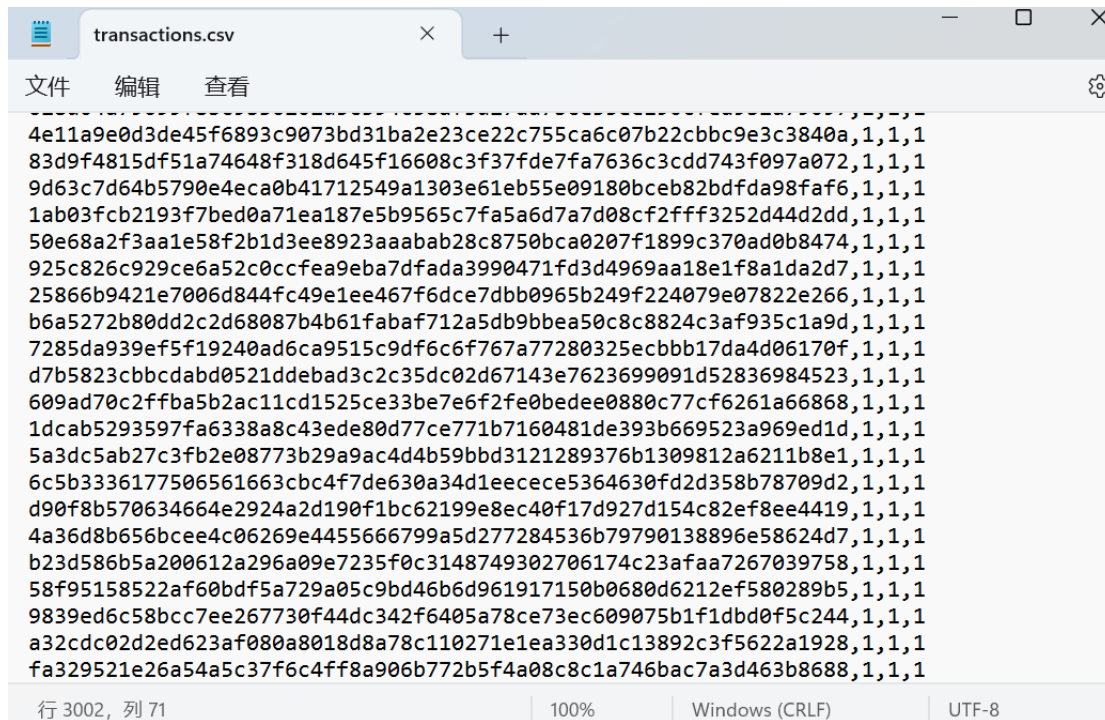
Information:
The length of the block chain is: 0
The number of transactions: 0
```

这时由于 Node1 还没中奖过，所以区块长度和交易次数都是 0。在处理 Node2 信息之前我们先看看交易池里的内容，方便对比。

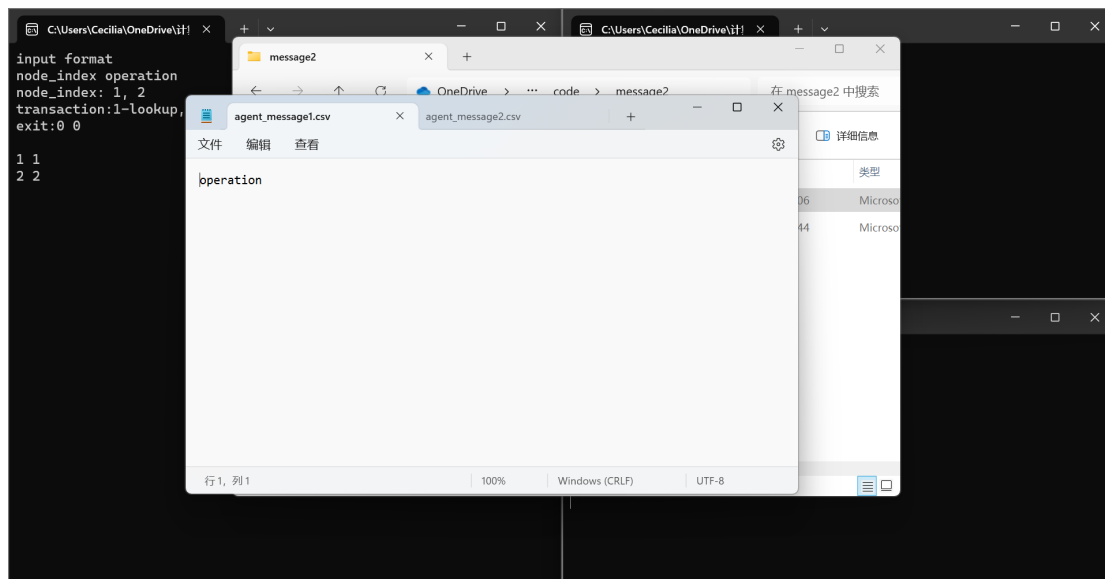
```
transactions.csv x + - □ x
文件 编辑 查看
-----
4e11a9e0d3de45f6893c9073bd31ba2e23ce22c755ca6c07b22cbbc9e3c3840a,1,1,1
83d9f4815df51a74648f318d645f16608c3f37fde7fa7636c3cdd743f097a072,1,1,1
9d63c7d64b5790e4eca0b41712549a1303e61eb55e09180bceb82bdfda98faf6,1,1,1
1ab03fcb2193f7bed0a71ea187e5b9565c7fa5a6d7a7d08cf2fff3252d44d2dd,1,1,1
50e68a2f3aa1e58f2b1d3ee8923aaabab28c8750bca0207f1899c370ad0b8474,1,1,1
925c826c929ce6a52c0ccfea9eba7dfada3990471fd3d4969aa18e1f8a1da2d7,1,1,1
25866b9421e7006d844fc49e1ee467f6dce7dbb0965b249f224079e07822e266,1,1,1
b6a5272b80dd2c2d68087b4b61fabaf712a5db9bbea50c8c8824c3af935c1a9d,1,1,1
7285da939ef5f19240ad6ca9515c9df6c6f767a77280325ecbbb17da4d06170f,1,1,1
d7b5823cbbcdabd0521ddeb3c2c35dc02d67143e7623699091d52836984523,1,1,1
609ad70c2ffba5b2ac11cd1525ce33be7e6f2fe0bedee0880c77cf6261a66868,1,1,1
1dcab5293597fa6338a8c43ede80d77ce771b7160481de393b669523a969ed1d,1,1,1
5a3dc5ab27c3fb2e08773b29a9ac4d4b59bbd3121289376b1309812a6211b8e1,1,1,1
6c5b3336177506561663cbc4f7de630a34d1eecece5364630fd2d358b78709d2,1,1,1
d90f8b570634664e2924a2d190f1bc62199e8ec40f17d927d154c82ef8ee4419,1,1,1
4a36d8b656bcee4c06269e4455666799a5d277284536b79790138896e58624d7,1,1,1
b23d586b5a200612a296a09e7235f0c3148749302706174c23afaa7267039758,1,1,1
58f95158522af60bdf5a729a05c9bd46b6d961917150b0680d6212ef580289b5,1,1,1
9839ed6c58bcc7ee267730f44dc342f6405a78ce73ec609075b1f1dbd0f5c244,1,1,1
a32cdc02d2ed623af080a8018d8a78c110271e1ea330d1c13892c3f5622a1928,1,1,1

行 1, 列 1 | 100% | Windows (CRLF) | UTF-8
```

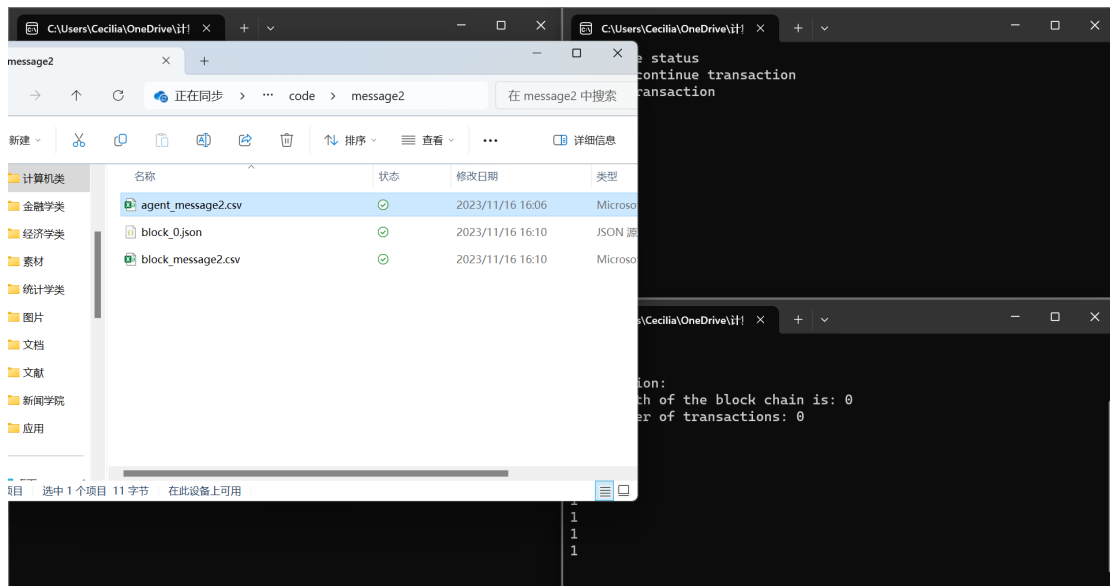
为了使交易信息更丰富，我保留了 is\_coinbase、input\_count、output\_count 等信息。可以看到队尾的交易 txid 是“a32...1928”，此时我们处理 Node2 的信息。



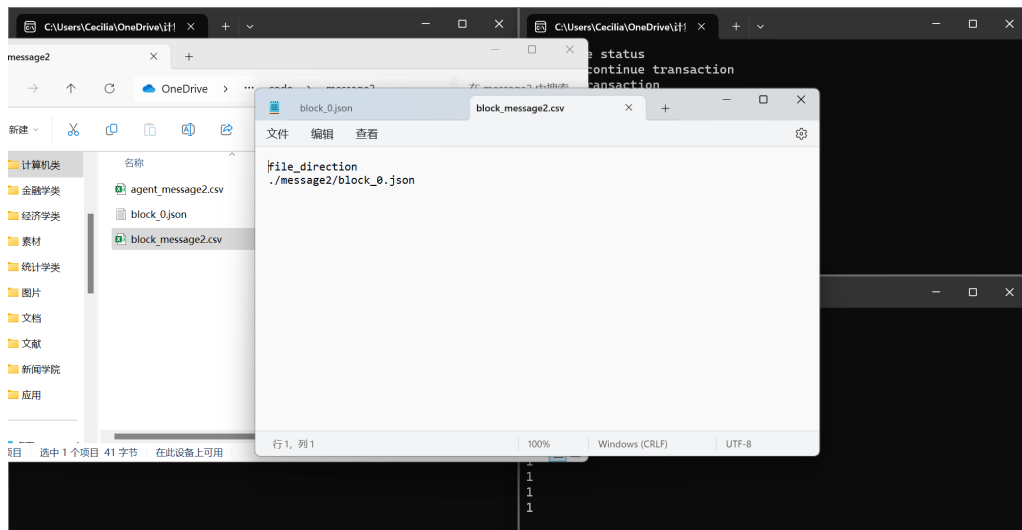
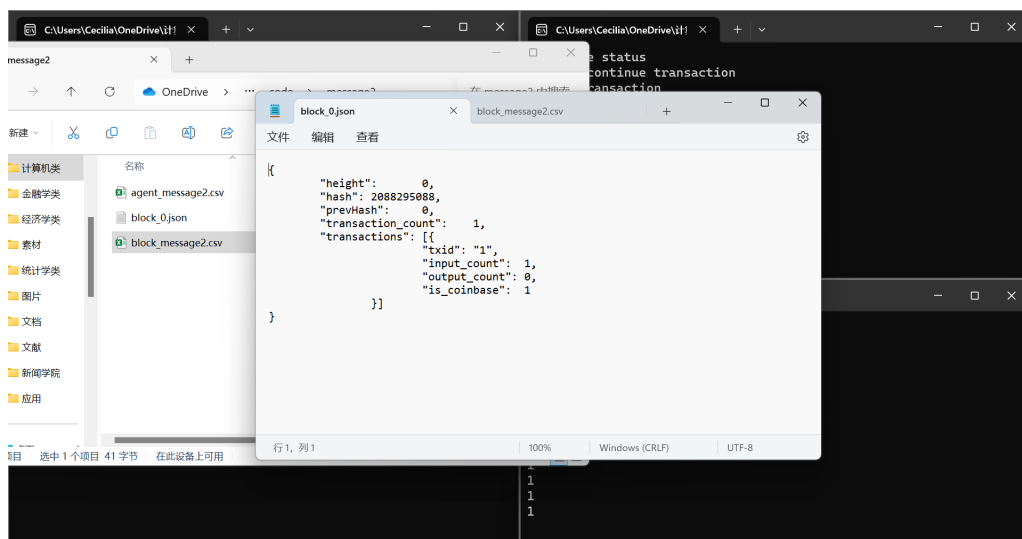
可以看到已经把新生成的 transaction 插入了队尾，此时再检查 agent\_message（客户消息队列）里的内容发现使用过的信息已经被清除。



接下来我们多运行几次 Node1 直到其中奖。



区块消息队列和用户消息队列都为空时，显然程序运行没有对各个文件夹做任何操作，说明程序是稳定的。经过多次努力 Node1 终于中奖了，使用交易池里的信息生成了区块，并将区块的地址存入了 `block_message`（区块消息队列），看看这个区块的信息。

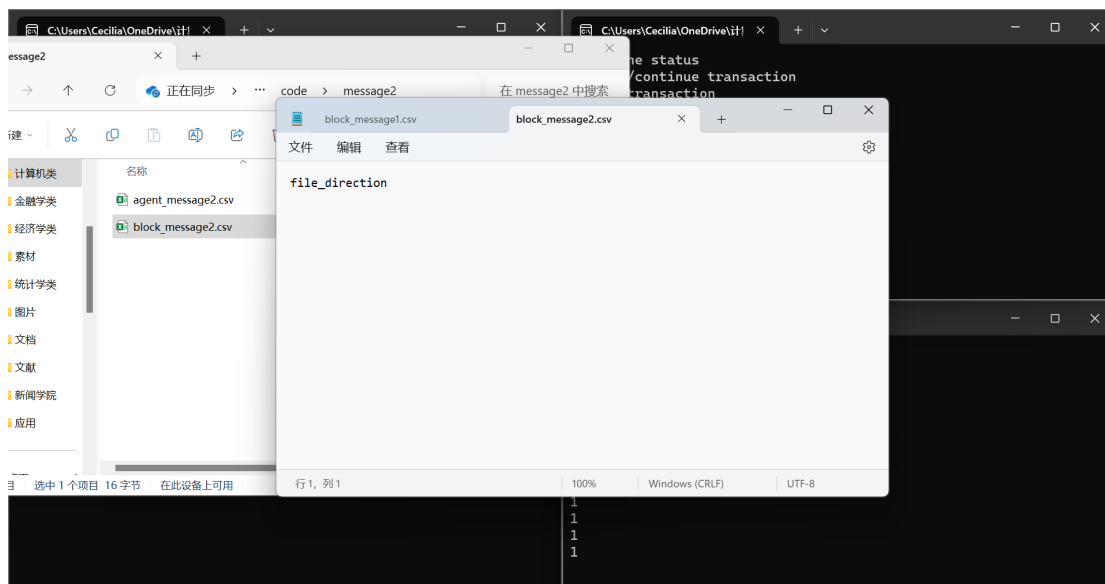




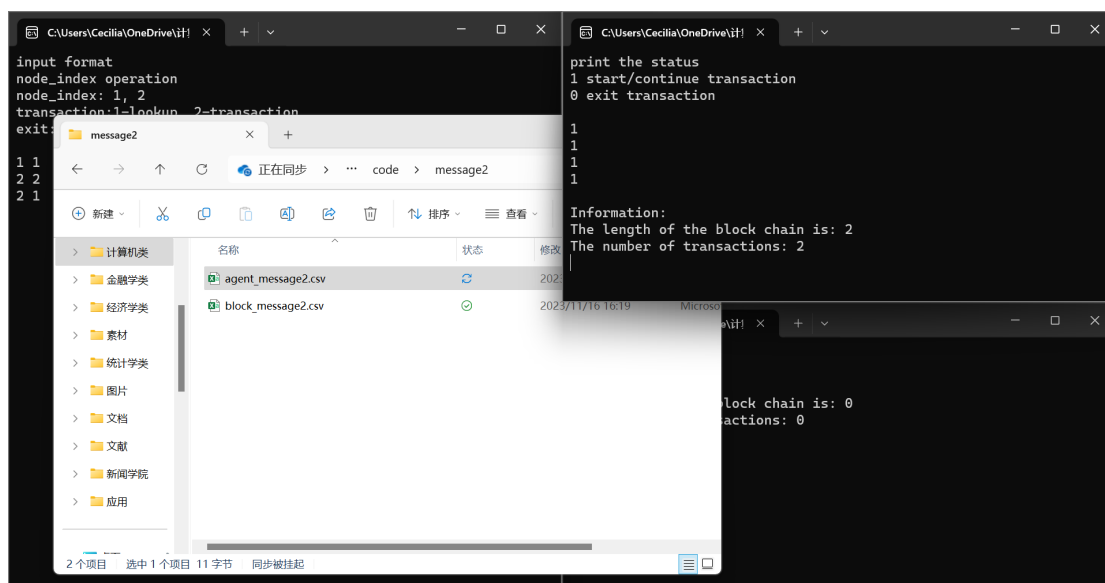
Node1 生成的地址和区块信息都成功写入 Node2 维护的文件夹，说明程序没有问题。

接下来让 Node2 运行一次，如果没中奖的话会处理 Node1 发送的区块消息队列。

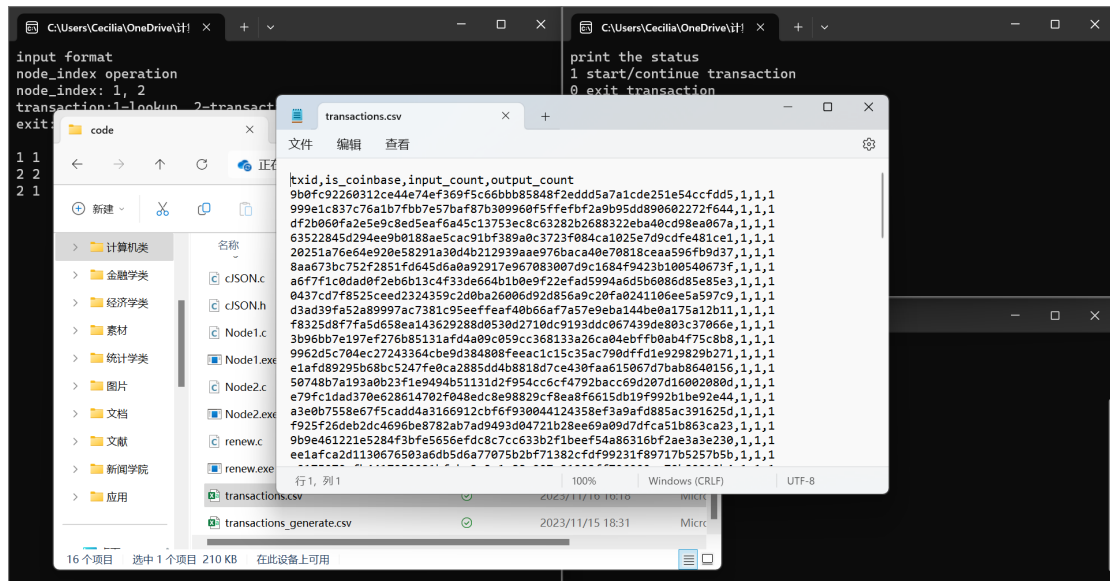
（运行了一次没反应我还以为出 bug 了结果发现是 Node2 中奖了，那再运行一次）



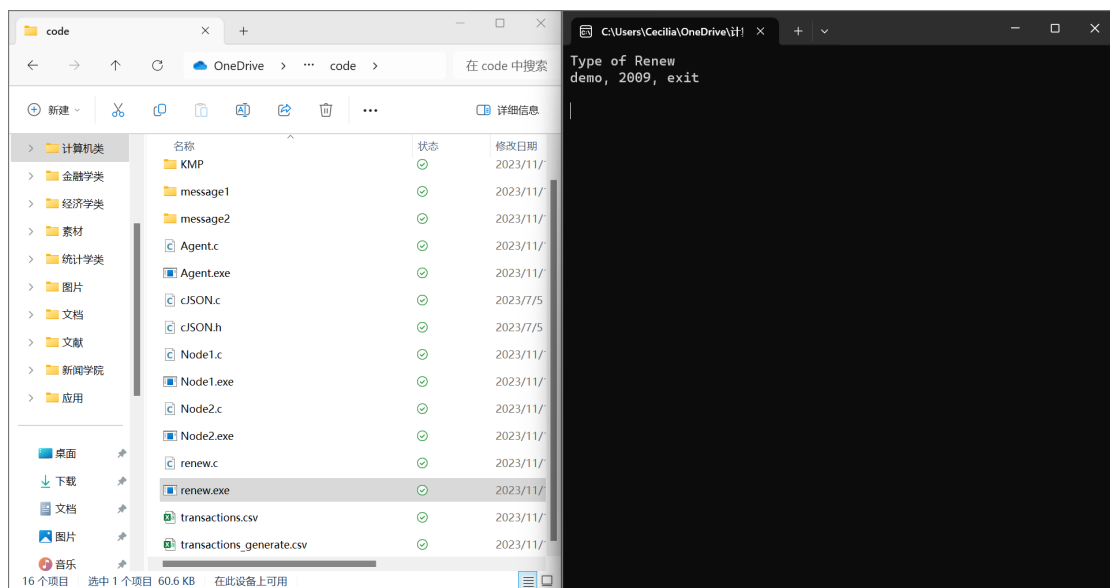
Node2 已经成功处理消息，并删除了消息，为了验证 Node2 是不是像想象中那样处理了消息，我们再使用 Agent 向 Node2 发送查询请求。



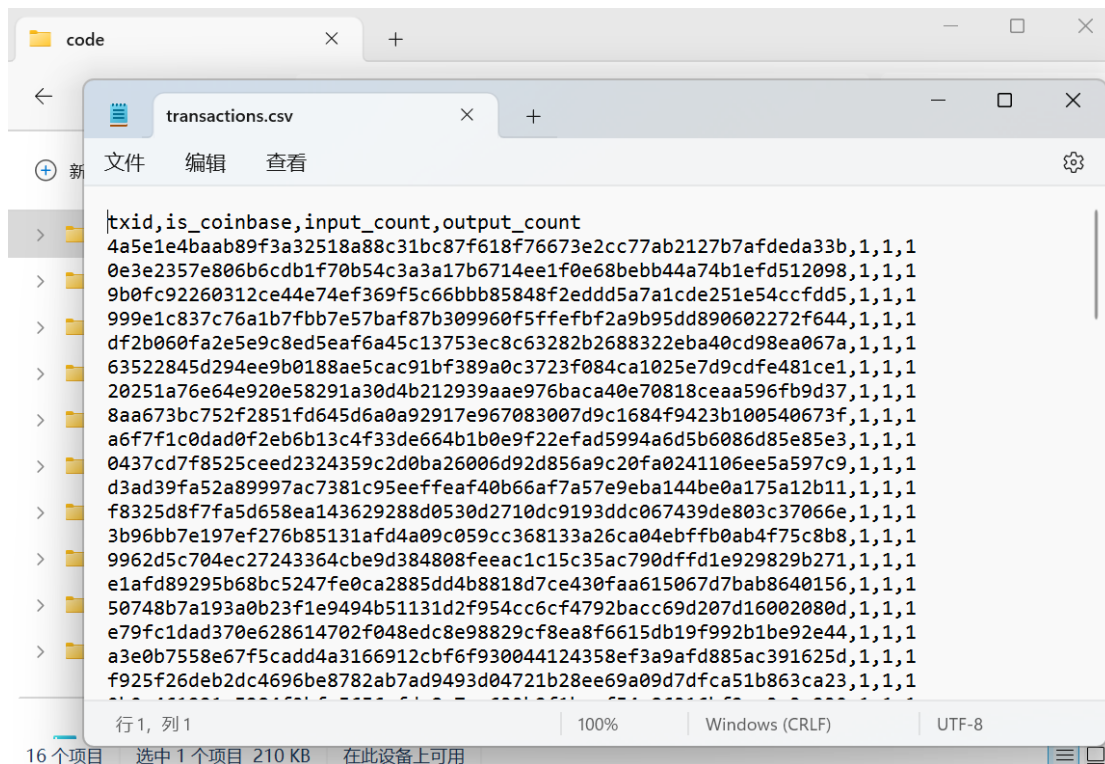
Node2 区块链长度是 2，对应自己中奖的那个区块和处理的 Node1 发送的那个区块，这时我们再打开 transactions.csv，发现用过且通过验证的交易信息已经被清除了。（好像忘记展示最开始 transactions.csv 是什么样了，待会 renew 再展示）



程序已经实现了实验要求的所有内容，接下来是我自己附加的一些内容。比如现在我们需要退出所有程序，Agent.exe 输入“0 0”即可退出，两个 Node 输入“0”即可退出，我就不演示了。接下来主要是演示 renew.exe，即我们刚刚使用这三个 exe 时改变了文件内容，现在我们要把文件复原。



我提示了三种操作，一种是用 demo 的交易去更新文件，另一种是用 2009 的交易去更新文件，或者直接退出（更新后会自动退出），这里我们用 2009 的信息去更新文件。



这时我们发现 transactions.csv（交易池）已经恢复如初了，至此区块链的相关内容已经展示结束，如果需要探索其他功能可以自行探索，相关文件我已经传送到 OBE 上，同时在 github 上直接搜索 mini\_blockchain 也可找到。（如果没找到就是我还没传，迟早会传的）

### 接下来是 KMP 算法的演示

由于我确实没毛病 KMP 和我们的实验有什么关系，我就单独写了一个 KMP 算法。首先我写了三个函数，分别用于计算 next、nextval 和 KMP 算法得到的 index。

```
void computeNext(char *pattern, int next[]);
void computeNextVal(char *pattern, int nextval[]);
int KMP(char *text, char *pattern, int next[]);
```

我们用一段文本来演示一下。

```
int main()
{
    char text[] = "AABABCABABCABABC";
    char pattern[] = "ABCABABC";

    int m = strlen(pattern);
    int next[m];
    int nextval[m];

    computeNext(pattern, next);
    computeNextVal(pattern, nextval);
```

```

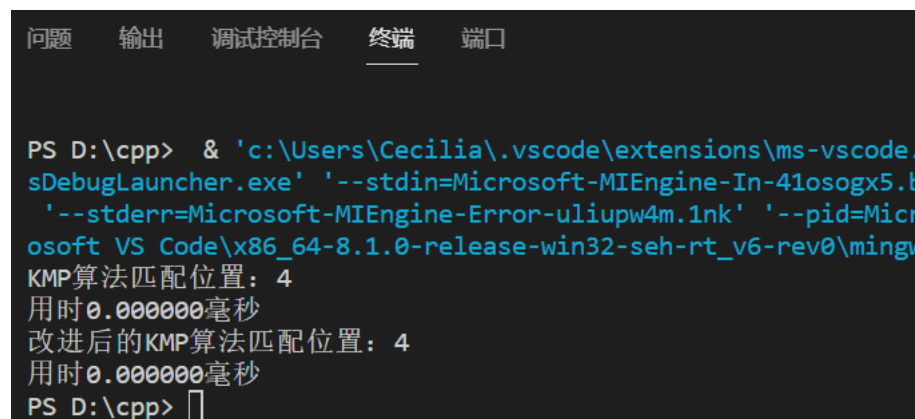
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    int result = KMP(text, pattern, next);
    end = clock();
    cpu_time_used = (double)(end - start);
    printf("KMP 算法匹配位置: %d\n 用时%lf 毫秒\n", result, cpu_time_used);

    start = clock();
    result = KMP(text, pattern, nextval);
    end = clock();
    cpu_time_used = (double)(end - start);
    printf("改进后的 KMP 算法匹配位置: %d\n 用时%lf 毫秒\n", result,
cpu_time_used);

    return 0;
}

```

为了探寻那种算法更快我专门给每个算法计时作为比较，运行结果如下：



```

问题  输出  调试控制台  终端  端口

PS D:\cpp> & 'c:\Users\Cecilia\.vscode\extensions\ms-vscode
sDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-41osogx5.1
'--stderr=Microsoft-MIEngine-Error-uliupw4m.1nk' '--pid=Micro
soft VS Code\x86_64-8.1.0-release-win32-seh-rt_v6-rev0\mingw
KMP算法匹配位置: 4
用时0.000000毫秒
改进后的KMP算法匹配位置: 4
用时0.000000毫秒
PS D:\cpp> 

```

好吧可能计时上面有一小问题，但两种算法得到了一样的结果。

## 5. 遇到的问题和解决方案

- 包含头文件时 VScode 无法编译文件：使用 gcc 直接编译，然后用 gdb 手动 debug，例如编译和运行 Node1.c 时：cd ./lab4; gcc Node1.c cJSON.c -o Node1.exe; gdb Node1.exe; c; r。
- 很难用文件队列存储区块信息：用一个 csv 队列存储区块.json 文件的地址。
- C 语言的生成的随机数为伪随机数：用当前的时间信息生成随机数种子 srand((unsigned int)time(NULL))。

- 无法在 VScode 里面同时运行多个程序：但是可以在 windows 系统里同时运行多个 exe，这带来的唯一麻烦就是不太好 debug。
- 每次运行过后重新放文件比较麻烦：backup 和 renew.exe 可以帮助自动更新文件。