实验课题：死锁检测算法模拟实现

计科 1903 班　　19281171　王雨潇

1. 实验目的

理解并掌握死锁检测算法的基本设计思想、关键数据结构和算法流程。

2. 实验环境

运行环境：Windows 10 操作系统

编译器：MinGW-w64 GCC 11.2.0

IDE：VSCode，实验代码见“附录：源程序的完整代码”，所有流程图均使用 plantuml 绘制

3. 实验内容

利用 C 语言设计与实现死锁检测算法，构建计算机系统的进程-资源场景随机发生机制，并对自己的死锁检测算法实现方案加以测试验证。

3.1. 关键数据结构定义

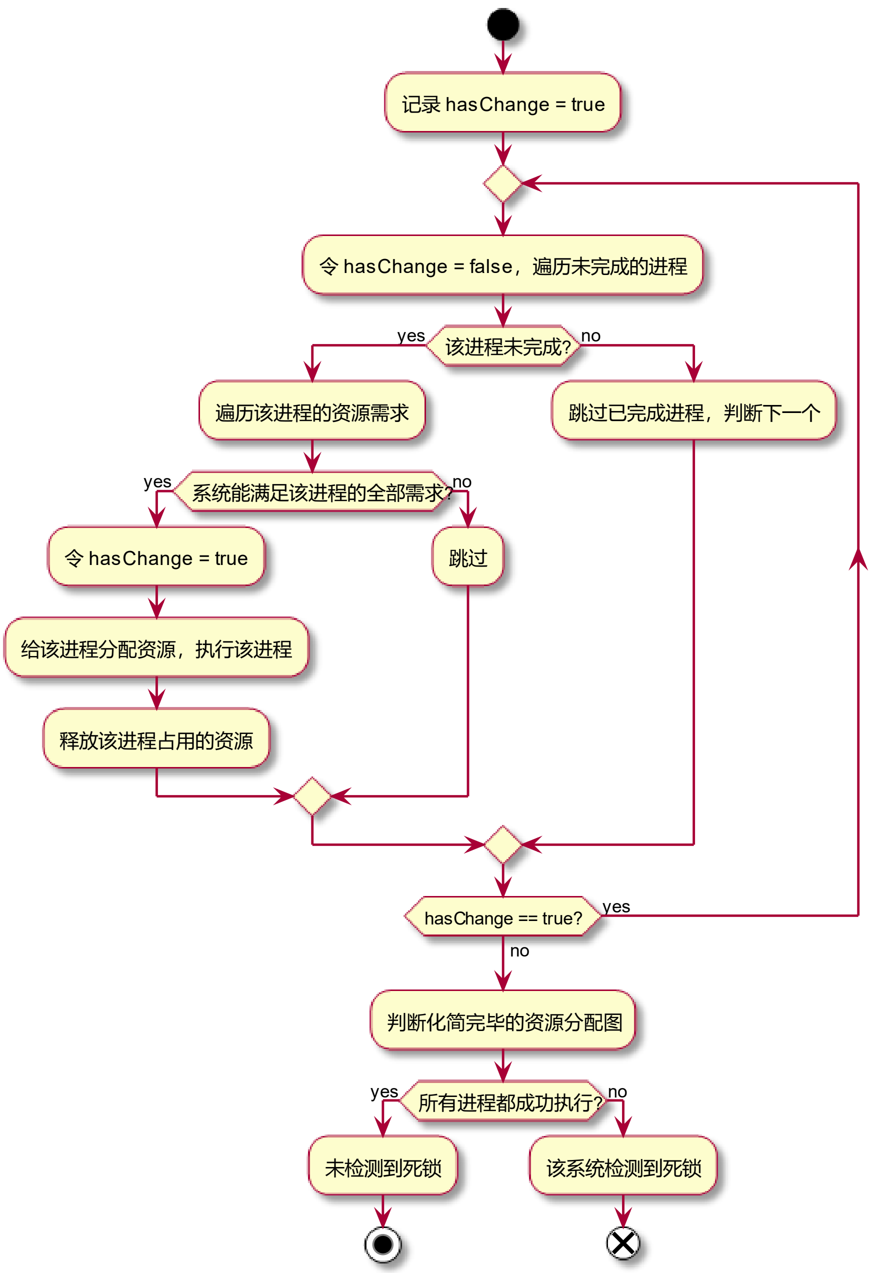
如图所示，定义结构体 struct scene 用于描述一组系统资源和一组进程请求的关系；

```c
#define bool    int       // 假装有 bool 类型
#define true    1
#define false   0
#define author  19281171  // 署个名

// 结构体 scene 用于描述一组系统资源进程分配场景
typedef struct {
    int n;               // 进程数量为 n
    int m;               // 资源种类为 m
    int *amount;         // 可用资源的数量 int[m]
    int *available;      // 系统提供的各类资源的数目 int[m]
    int **request;       // 进程当前对各类资源的请求数目 int[n][m]
    int **allocation;    // 当前时刻的资源的分配情况 int[n][m]
    bool *hasFinished;   // 进程是否完成 bool[n]
} scene;
```

3.2. 死锁检测算法的实现

查阅教材定义可知，系统状态 S 为死锁状态的**充要条件**是当且仅当该状态下的资源分配图 RAG（以下简称为 RAG）是**不可完全化简**的。因此，本程序采用对 RAG 不断循环遍历的方式化简，化简顺序即进程执行次序，重复操作直到 RAG 不再产生变化，此时进程状态即进程执行的最终结果，若所有进程都是执行完毕的状态，则该系统没有发生死锁，反之则发生了死锁。

该死锁检测算法的流程图描述如下：

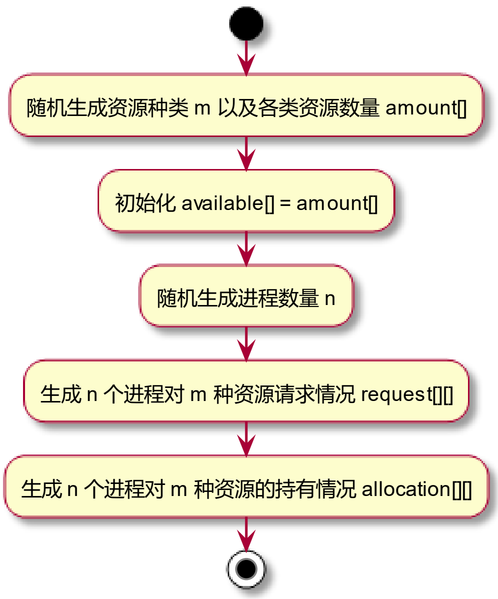


基于这一流程设计，可写出基于 C 语言的程序实现，关键循环体代码及注释如下：

```c
bool hasChange = true;
int timePast = 1;

while (hasChange) {
    // 化简资源分配图 RAG
    printf("Round %d: Analyzing RAG ... \n", timePast++);
    hasChange = false;
    for (int i=0; i < s->n; i++) {
        if (s->hasFinished[i] == false) {
            bool isSatisfied = true;
            // 若该进程还没完成，试判断系统能否满足该进程的全部需求
            for (int j=0; j < s->m; j++) {
                if ((s->request[i][j] - s->allocation[i][j]) > s->available[j]) {
                    isSatisfied = false;    // 不能满足该进程
                    break;
                }
            }

            if (isSatisfied) {
                hasChange = true;   // 若该进程的要求可以全部被满足，则进程可以完成
                s->hasFinished[i] = true;
                for (int j=0; j < s->m; j++) {
                    // 释放进程 i，资源返还
                    s->available[j] = s->available[j] + s->allocation[i][j];
                    s->allocation[i][j] = 0;
                }
            }
        }
    }
}

// 当 RAG 不再有新变化时，化简完毕退出循环，检测各进程的需求是否都得到满足
bool isSuccess = true;
for (int i=0; i < s->n; i++) {
    if (s->hasFinished[i] == false) { isSuccess = false; break; }
}
```

3.3. 随机发生计算机系统的进程-资源场景

本程序依据“3.1. 关键数据结构定义”使用 C 语言一维数组和二维数组模拟资源数量和占用情况，随机生成进程-资源场景样例，即用符合常理的合法随机数填充代表资源的数组。流程图如下所示：



具体代码实现可参考附录或附件的源码文件。

3.4. 基于检测算法判断死锁状态

调用测试函数，测试结果显示，算法原型能正确判断各种进程-资源场景是否陷入了死锁状态。

摘取其中两组运行结果如下：

测试用例 1：出现死锁

```
--------------------INIT-SCENE-SHOW--------------------
n=10
m=10
amount=[16, 29, 3, 20, 29, 12, 7, 4, 26, 29]
available=[6, 6, 3, 7, 10, 7, 5, 4, 6, 8]
request=[[1, 27, 0, 13, 27, 11, 2, 2, 1, 12],
       [8, 2, 1, 4, 28, 0, 6, 0, 14, 14],
       [3, 1, 0, 13, 10, 10, 4, 0, 18, 22],
       [0, 9, 1, 17, 18, 5, 6, 2, 12, 12],
       [6, 8, 1, 13, 17, 7, 6, 3, 6, 14],
       [8, 4, 2, 5, 28, 4, 6, 0, 23, 18],
       [2, 27, 0, 16, 0, 8, 1, 0, 20, 6],
       [5, 0, 0, 4, 25, 0, 4, 2, 7, 28],
       [15, 16, 0, 10, 13, 3, 3, 1, 24, 7],
```

```
       [0, 19, 0, 16, 10, 7, 4, 0, 1, 14]]
allocation=[[2, 3, 0, 2, 3, 0, 0, 0, 7, 2],
       [2, 4, 0, 5, 1, 1, 1, 0, 2, 0],
       [2, 0, 0, 2, 6, 0, 1, 0, 2, 7],
       [0, 6, 0, 1, 0, 2, 0, 0, 3, 5],
       [2, 2, 0, 0, 5, 0, 0, 0, 1, 2],
       [1, 3, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 2, 0, 0, 1, 0, 0, 0, 2, 3],
       [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 1, 0, 1, 2, 1, 0, 0, 1, 2],
       [0, 1, 0, 2, 1, 0, 0, 0, 1, 0]]
hasFinished=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Round 1: Analyzing RAG ...

---------------------END-SCENE-SHOW---------------------
n=10
m=10
amount=[16, 29, 3, 20, 29, 12, 7, 4, 26, 29]
available=[6, 6, 3, 7, 10, 7, 5, 4, 6, 8]
request=[[1, 27, 0, 13, 27, 11, 2, 2, 1, 12],
       [8, 2, 1, 4, 28, 0, 6, 0, 14, 14],
       [3, 1, 0, 13, 10, 10, 4, 0, 18, 22],
       [0, 9, 1, 17, 18, 5, 6, 2, 12, 12],
       [6, 8, 1, 13, 17, 7, 6, 3, 6, 14],
       [8, 4, 2, 5, 28, 4, 6, 0, 23, 18],
       [2, 27, 0, 16, 0, 8, 1, 0, 20, 6],
       [5, 0, 0, 4, 25, 0, 4, 2, 7, 28],
       [15, 16, 0, 10, 13, 3, 3, 1, 24, 7],
       [0, 19, 0, 16, 10, 7, 4, 0, 1, 14]]
allocation=[[2, 3, 0, 2, 3, 0, 0, 0, 7, 2],
       [2, 4, 0, 5, 1, 1, 1, 0, 2, 0],
       [2, 0, 0, 2, 6, 0, 1, 0, 2, 7],
       [0, 6, 0, 1, 0, 2, 0, 0, 3, 5],
       [2, 2, 0, 0, 5, 0, 0, 0, 1, 2],
       [1, 3, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 2, 0, 0, 1, 0, 0, 0, 2, 3],
       [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 1, 0, 1, 2, 1, 0, 0, 1, 2],
       [0, 1, 0, 2, 1, 0, 0, 0, 1, 0]]
hasFinished=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]


Error: The system has deadlock!
```

测试用例 2：没有发生死锁

```
---------------------INIT-SCENE-SHOW---------------------
n=9
m=9
amount=[20, 0, 17, 11, 17, 6, 0, 8, 9]
available=[9, 0, 8, 7, 8, 5, 0, 5, 5]
request=[[0, 0, 8, 7, 8, 4, 0, 0, 2],
       [11, 0, 3, 5, 11, 2, 0, 6, 1],
       [12, 0, 15, 2, 9, 3, 0, 2, 0],
       [15, 0, 2, 5, 16, 3, 0, 1, 3],
       [2, 0, 8, 10, 8, 4, 0, 2, 4],
```

```
       [14, 0, 9, 10, 14, 5, 0, 3, 3],
       [14, 0, 15, 7, 9, 5, 0, 3, 8],
       [0, 0, 14, 6, 4, 5, 0, 1, 6],
       [3, 0, 12, 9, 4, 2, 0, 6, 4]]
allocation=[[3, 0, 0, 2, 3, 1, 0, 0, 1],
       [0, 0, 3, 1, 3, 0, 0, 1, 1],
       [1, 0, 0, 0, 2, 0, 0, 1, 0],
       [2, 0, 2, 0, 0, 0, 0, 0, 1],
       [0, 0, 2, 0, 0, 0, 0, 0, 1],
       [2, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 0, 1, 1, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [2, 0, 2, 0, 0, 0, 0, 0, 0]]
hasFinished=[0, 0, 0, 0, 0, 0, 0, 0, 0]
Round 1: Analyzing RAG ...
Round 2: Analyzing RAG ...
Round 3: Analyzing RAG ...

---------------------END-SCENE-SHOW---------------------
n=9
m=9
amount=[20, 0, 17, 11, 17, 6, 0, 8, 9]
available=[20, 0, 17, 11, 17, 6, 0, 8, 9]
request=[[0, 0, 8, 7, 8, 4, 0, 0, 2],
       [11, 0, 3, 5, 11, 2, 0, 6, 1],
       [12, 0, 15, 2, 9, 3, 0, 2, 0],
       [15, 0, 2, 5, 16, 3, 0, 1, 3],
       [2, 0, 8, 10, 8, 4, 0, 2, 4],
       [14, 0, 9, 10, 14, 5, 0, 3, 3],
       [14, 0, 15, 7, 9, 5, 0, 3, 8],
       [0, 0, 14, 6, 4, 5, 0, 1, 6],
       [3, 0, 12, 9, 4, 2, 0, 6, 4]]
allocation=[[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0]]
hasFinished=[1, 1, 1, 1, 1, 1, 1, 1, 1]


Successed! :)
```

4. 实验结论和心得体会

在本次实验中，我遇到的主要困难有两个，其一是不理解 RAG 图化简如何用代码来表达（后来发现不一定要还原图结构），其二是不知道随机生成场景时，如何设计才能生成比较能全面体现检测能力的用例。通过本次实验，我了解了死锁检测算法的基本设计思想，在实践中通过设计数据结构和算法流程加深了自己对死锁概念和判断的认知，也对检测死锁有了更深入的理解和认识。

5. 附录：源程序的完整代码

```
#include <bits/stdc++.h>

using namespace std;

#define bool    int       // 假装有 bool 类型
#define true    1
#define false   0
#define author  19281171  // 署个名

// 结构体 scene 用于描述一组系统资源进程分配场景
typedef struct {
    int n;                // 进程数量为 n
    int m;                // 资源种类为 m
    int *amount;          // 可用资源的数量 int[m]
    int *available;       // 系统提供的各类资源的数目 int[m]
    int **request;        // 进程当前对各类资源的请求数目 int[n][m]
    int **allocation;     // 当前时刻的资源的分配情况 int[n][m]
    bool *hasFinished;    // 进程是否完成 bool[n]
} scene;

// 展示 scene 结构体的值
void DisplayScene(scene *s) {
    printf("n=%d\n", s->n);
    printf("m=%d\n", s->m);

    printf("amount=[");
    for(int i=0; i < s->m; i++) {
        if (i != s->m - 1) printf("%d, ", s->amount[i]);
        else printf("%d]\n", s->amount[i]);
    }

    printf("available=[");
    for(int i=0; i < s->m; i++) {
        if (i != s->m - 1) printf("%d, ", s->available[i]);
        else printf("%d]\n", s->available[i]);
    }

    printf("request=[[");
    for (int i=0; i < s->n; i++) {
        if (i != 0) printf("\t[");
```

```
        for (int j=0; j < s->m; j++) {
            if (j != s->m - 1) printf("%d, ", s->request[i][j]);
            else printf("%d]", s->request[i][j]);
        }
        if (i != s->n - 1) printf(",\n");
        else printf("]\n");
    }

    printf("allocation=[[");
    for (int i=0; i < s->n; i++) {
        if (i != 0) printf("\t[");
        for (int j=0; j < s->m; j++) {
            if (j != s->m - 1) printf("%d, ", s->allocation[i][j]);
            else printf("%d]", s->allocation[i][j]);
        }
        if (i != s->n - 1) printf(",\n");
        else printf("]\n");
    }

    printf("hasFinished=[");
    for(int i=0; i < s->n; i++) {
        if (i != s->n - 1) printf("%d, ", s->hasFinished[i]);
        else printf("%d]\n", s->hasFinished[i]);
    }
}

// 随机生成系统各类资源配备数量明细、一组并发进程、当前各类资源占有/需求数量
scene *GenerateRandomScene() {
    scene *s = (scene *)malloc(sizeof(scene));
    s->m = rand() % (10 - 5 + 1) + 5;                    // 随机资源种类 [5, 10]
    s->amount = (int *)malloc(sizeof(int) * (s->m));    // 开 amount 数组的空间
    for (int i=0; i < s->m; i++) {
        s->amount[i] = rand() % 31;                     // 生成随机各类资源数量 [0, 30]
    }

    s->available = (int *)malloc(sizeof(int) * (s->m)); // 开 available 数组的空间
    for (int i=0; i < s->m; i++) {
        s->available[i] = s->amount[i];                 // 初始化 available = amount
    }

    s->n = rand() % (10 - 5 + 1) + 5;                    // 随机进程数量 [5, 10]
    s->hasFinished = (bool *)malloc(sizeof(bool) * (s->n)); // 开 hasFinished 数组空间
    for (int i=0; i < s->n; i++) {
        s->hasFinished[i] = false;                      // 初始状态所有进程均未完成
    }

    // 开 request 数组的 n 行 m 列空间
    s->request = (int **)malloc(sizeof(int *) * (s->n));
    for (int i=0; i < s->n; i++) {
        // 初始化各进程的资源请求数
        s->request[i] = (int *)malloc(sizeof(int) * (s->m));
        for (int j=0; j < s->m; j++) {
            // 初始化该进程对每种类型资源的请求数，随机为 [0, 该资源总数]
            s->request[i][j] = (s->amount[j] != 0)? rand() % (s->amount[j]): 0;
        }
    }

    // 开 allocation 数组的 n 行 m 列空间
```

```c
    s->allocation = (int **)malloc(sizeof(int *) * (s->n));
    for (int i=0; i < s->n; i++) {
        s->allocation[i] = (int *)malloc(sizeof(int) * (s->m));
        for (int j=0; j < s->m; j++) {
            // 初始化该进程已取得每种类型的资源数，随机为 [0, 可用资源数/3]
            int randAlloc = (s->available[j]/3 != 0)? rand() % (s->available[j]/3): 0;
            // 从 available 数组中减去分配的资源
            s->allocation[i][j] = randAlloc;
            s->available[j] -= randAlloc;
        }
    }

    return s;
}

// 检测该场景是否存在死锁
bool isDeadLock(scene *s) {
    printf("\n--------------------INIT-SCENE-SHOW--------------------\n");
    DisplayScene(s);
    bool hasChange = true;
    int timePast = 1;

    while (hasChange) {
        // 化简资源分配图 RAG
        printf("Round %d: Analyzing RAG ... \n", timePast++);
        hasChange = false;
        for (int i=0; i < s->n; i++) {
            if (s->hasFinished[i] == false) {
                bool isSatisfied = true;
                // 若该进程还没完成，试判断系统能否满足该进程的全部要求
                for (int j=0; j < s->m; j++) {
                    if ((s->request[i][j] - s->allocation[i][j]) > s->available[j]) {
                        isSatisfied = false;    // 不能满足该进程
                        break;
                    }
                }

                if (isSatisfied) {
                    hasChange = true;   // 若该进程的要求可以全部被满足，则进程可以完成
                    s->hasFinished[i] = true;
                    for (int j=0; j < s->m; j++) {
                        // 释放进程 i, 资源返还
                        s->available[j] = s->available[j] + s->allocation[i][j];
                        s->allocation[i][j] = 0;
                    }
                }
            }
        }
    }

    // 当 RAG 不再有新变化时，化简完毕退出循环，检测进程的需求是否都得到满足
    bool isSuccess = true;
    for (int i=0; i < s->n; i++) {
        if (s->hasFinished[i] == false) { isSuccess = false; break; }
    }

    printf("\n--------------------END-SCENE-SHOW--------------------\n");
    DisplayScene(s);
```

```c
    printf("\n\n");

    if (isSuccess) {
        printf("Successed! :) \n");
        return false;
    } else {
        printf("Error: The system has deadlock!\n");
        return true;
    }
}

int main() {
    srand((unsigned)time(NULL));    // 初始化随机数种子
    int n = 10;
    for (int i=0; i<n; i++) {        // 一次生成 n 组用例
        scene *s = GenerateRandomScene();
        isDeadLock(s);              // 判断是否为死锁
        free(s);                    // 释放结构体空间
    }

    return 0;
}
```