Sudoku

목차

컴파일 과정

```
parkminseon@bagminseon-ui-MacBookPro proj2-2 % make
gcc -Wall -O -c sudoku.c
gcc -o sudoku sudoku.o
parkminseon@bagminseon-ui-MacBookPro proj2-2 %
```

기능 설명

1. `check_rows` 함수

check_rows 함수는 스도쿠 표에서 각각의 행이 모두 올바르게 구성되어있는지 검사하는 함수입니다. 즉, 각 행에서 1부터 9까지의 값이 중복 없이 올바르게 포함되어 있는지 검사해야 합니다.

```c
/*
 * 스도쿠 퍼즐의 각 행이 올바른지 검사한다.
 * 행 번호는 0부터 시작하며, i번 행이 올바르면 valid[0][i]에 true를 기록한다.
 */
void *check_rows(void *arg)
{
    /*
     * 0번 행부터 8번 행까지 순차적으로 검사한다.
     * 각 행은 또 다시 0번 ~ 8번 열로 구성되어 있으므로, 2중 for문을 사용하여 각 값에 접근한다.
     */
    for (int row = 0; row < 9; ++row) {
        /*
         * isVisited 배열은 인덱스에 해당하는 숫자가 현재 행에서 등장했는지 체크한다.
         * 단, 자연수 그대로를 인덱스로 사용하여 0번 인덱스는 사용하지 않는다.
         */
        bool isVisited[10] = {0};

        for (int column = 0; column < 9; ++column) {
            int current = sudoku[row][column];

            /*
             * 행 안에서 값이 중복하여 등장하는 경우
             * 해당 행이 올바르지 않음을 표시하고 해당 행의 검사를 중단한다.
             */
            if (isVisited[current]) {
                valid[0][row] = false;
                break; // 다음 행을 검사한다.
            }

            isVisited[current] = true;
            /*
             * 마지막 열에 대한 검사까지 성공적으로 마쳤다면, 행이 올바르게 구성되었음을 기록한다.
             */
            if (column == 8) valid[0][row] = true;
        }
    }
    pthread_exit(NULL);
}
```

우선 ***2차원 배열*** 형태의 스도쿠에 저장된 값에 순차적으로 하나씩 접근해야 하므로 **중첩 for문**을 사용하였습니다. 그리고 **행 단위**로 검사를 진행하기 때문에 행 번호에 대한 for문을 외부에, 열 번호에 대한 for문은 내부에 배치하였습니다.

그리고 각 행에서 1부터 9까지의 정수가 중복해서 등장했는지 검사하기 위해 하나의 행을 검사할 때마다 `isVisited` 배열을 생성하였습니다. `isVisited` 배열은 **인덱스에 해당하는 값이 현재 행에서 등장했는지의 여부**를 저장하는 배열입니다.

> 예를 들어 현재 검사하고 있는 행에서 숫자 **8**을 만나게 되면, `isVisited[8]`을 `true` 로 변경하는 방식입니다. 만약 이 과정에서 ***8이 이미 방문처리*** 된 상태라면 8이 중복해서 등장한 것이므로, 이 경우에는 ***행이 올바르지 않은 경우***입니다.

만약 `isVisited` 검사를 통해 현재 행이 올바르지 않다고 판단되면, 현재 행의 검사 결과로 `false`를 기록한 후 바로 다음 행부터 이어서 검사하기 위해 `break` 문으로 흐름을 제어하였습니다. 만약 마지막 열까지 검사가 올바르게 진행되었다면 현재 행의 검사 결과로 `true`를 기록하였습니다.

마지막으로 모든 행에 대한 검사를 마치면 해당 스레드를 종료하도록 `pthread_exit` 함수를 호출하였습니다.

2. `check_columns` 함수

check_columns 함수는 스도쿠 표에서 각각의 열이 모두 올바르게 구성되어있는지 검사하는 함수입니다. 즉, 각 열에서 1부터 9까지의 값이 중복 없이 올바르게 포함되어 있는지 검사해야 합니다.

```c
/*
 * 스도쿠 퍼즐의 각 열이 올바른지 검사한다.
 * 열 번호는 0부터 시작하며, j번 열이 올바르면 valid[1][j]에 true를 기록한다.
 */
void *check_columns(void *arg)
{
    /*
     * 0번 열부터 8번 열까지 순차적으로 탐색한다.
     * 각 열은 또 다시 0번 ~ 8번 행으로 구성되어 있으므로, 2중 for문을 사용하여 각 값에 접근한다.
     */
    for (int column = 0; column < 9; ++column) {
        /*
         * isVisited 배열은 인덱스에 해당하는 숫자가 현재 열에서 등장했는지 체크한다.
         * 단, 자연수 그대로를 인덱스로 사용하여 0번 인덱스는 사용하지 않는다.
         */
        bool isVisited[10] = {0};

        for (int row = 0; row < 9; ++row) {
            int current = sudoku[row][column];

            /*
             * 열 안에서 값이 중복하여 등장하는 경우
             * 해당 열이 올바르지 않음을 표시하고 해당 열의 검사를 중단한다.
             */
            if (isVisited[current]) {
                valid[1][column] = false;
                break; // 다음 열을 검사한다.
            }

            isVisited[current] = true;
            /*
             * 마지막 행에 대한 검사까지 성공적으로 마쳤다면, 열이 올바르게 구성되었음을 기록한다.
             */
            if (row == 8) valid[1][column] = true;
        }
    }
    pthread_exit(NULL);
}
```

check_rows 함수와 마찬가지로 **중첩 for문**을 사용하였습니다. 다만, 여기서는 **열 단위**로 검사를 진행하기 때문에 *check_rows 함수*와는 반대로 열 번호에 대한 for문을 외부에, 행 번호에 대한 for문을 내부에 배치하였습니다.

이후 검사 과정에 대한 코드는 `check_rows` 함수와 동일하게 작성하였으며, 여기서도 마찬가지로 `isVisited` 배열을 사용하여 열이 올바르게 구성되어 있는지 판단하였습니다.

3. `check_subgrid` 함수

check_subgrid 함수는 스도쿠 표에서 현재의 subgrid가 올바르게 구성되어있는지 검사하는 함수입니다. 즉, 현재의 subgrid에서 1부터 9까지의 값이 중복 없이 올바르게 포함되어 있는지 검사해야 합니다.

```c
/*
 * 스도쿠 퍼즐의 각 3x3 서브그리드가 올바른지 검사한다.
 * 3x3 서브그리드 번호(k)는 0부터 시작하며, 왼쪽에서 오른쪽, 위에서 아래로 증가한다. (0 ~ 8)
 * k번 서브그리드가 올바르면 valid[2][k]에 true를 기록한다.
 */
void *check_subgrid(void *arg)
{
    /*
     * 전달받은 arg를 int형의 k 변수로 변환한다.
     * k를 통해 시작 행 번호, 시작 열 번호를 계산한다.
     * 1 ~ 9까지의 숫자가 그리드 내에 등장했는지 확인하기 위해 isVisited 배열을 사용한다.
     */
    int k = *(int*)arg;
    int row_start = (k / 3) * 3;
    int col_start = (k % 3) * 3;
    bool isVisited[10] = {0};

    /*
     * k번째 3 X 3 서브 그리드를 검사한다.
     */
    for (int i = row_start; i < row_start + 3; ++i) {
        for (int j = col_start; j < col_start + 3; ++j) {
            int current = sudoku[i][j];

            /*
             * 숫자가 중복해서 등장한 경우,
             * k번 서브그리드에 대해 false 표시를 남기고 스레드를 종료한다.
             */
            if (isVisited[current]) {
                valid[2][k] = false;
                pthread_exit(NULL);
            }
            isVisited[current] = true;
        }
    }
    /*
     * 숫자가 중복해서 등장하지 않으면 for문이 정상적으로 종료되므로,
     * k번 서브그리드에 대해 true 표시를 남긴다.
     */
    valid[2][k] = true;
    pthread_exit(NULL);
}
```

check_subgrid 함수는 이전의 check_rows 함수와 check_columns 함수와는 다르게 subgrid 번호(K)에 대한 **인자값을 전달 받는다**는 차이점이 있습니다. 여기서 특히 주의할 점은 이 check_subgrid 함수는 추후 `check_sudoku` 함수에서 `pthread_create`를 통해 ***스레드로 생성***되기 때문에, 인자값을 포인터 값으로 전달 받는다는 점입니다. 따라서 check_subgrid의 인자 값이 `void pointer`로 설정되어 있는 것을 확인할 수 있습니다.

```c
    /*
     * 전달받은 arg를 int형의 k 변수로 변환한다.
     * k를 통해 시작 행 번호, 시작 열 번호를 계산한다.
     * 1 ~ 9까지의 숫자가 그리드 내에 등장했는지 확인하기 위해 isVisited 배열을 사용한다.
     */
    int k = *(int*)arg;
    int row_start = (k / 3) * 3;
    int col_start = (k % 3) * 3;
    bool isVisited[10] = {0};
```

여기서 인자로 전달 받은 **subgrid 번호(K)**를 통해 스도쿠 내에서 현재 subgrid의 범위가 어디부터 시작되는지 결정해야 합니다. 따라서 인자로 전달 받은 `void pointer`형의 arg값을 `int` 자료형으로 변환하여 변수 k에 할당하였습니다. 그리고 subgrid 번호(= `k`)와 시작

row 인덱스 / 시작 column 인덱스 사이의 관계식을 통해 `row_start` 변수와 `col_start` 변수에 시작 인덱스 값을 할당하였습니다.

```
121     /*
122      * k번째 3 X 3 서브 그리드를 검사한다.
123      */
124     for (int i = row_start; i < row_start + 3; ++i) {
125         for (int j = col_start; j < col_start + 3; ++j) {
126             int current = sudoku[i][j];
127 
128             /*
129              * 숫자가 중복해서 등장한 경우,
130              * k번 서브그리드에 대해 false 표시를 남기고 스레드를 종료한다.
131              */
132             if (isVisited[current]) {
133                 valid[2][k] = false;
134                 pthread_exit(NULL);
135             }
136             isVisited[current] = true;
137         }
138     }
139     /*
140      * 숫자가 중복해서 등장하지 않으면 for문이 정상적으로 종료되므로,
141      * k번 서브그리드에 대해 true 표시를 남긴다.
142      */
143     valid[2][k] = true;
144     pthread_exit(NULL);
```

이후 검사 과정은 앞선 check_rows 함수와 check_columns 함수와 유사하게 진행하였습니다. 다만 한 가지 차이점은 ***중복 값이 검사되는 순간 곧바로 스레드를 종료시킨다***는 점입니다. 앞선 check_rows와 check_columns는 모든 범위를 검사했다면, `check_subgrid`는 정해진 한 범위 내에서만 검사를 진행하기 때문에 중복값이 발견되는 순간 더이상 검사를 진행하지 않고 스레드를 곧바로 종료시킵니다.

4. 스레드 생성 및 조인

`check_sudoku` 함수는 위의 3개의 함수들을 활용하여 스도쿠를 검증하는 함수입니다.

```
165     /*
166      * 스레드를 생성하여 각 행을 검사하는 check_rows() 함수를 실행한다.
167      */
168     pthread_t row_thread;
169     pthread_create(&row_thread, NULL, check_rows, NULL);
170     /*
171      * 스레드를 생성하여 각 열을 검사하는 check_columns() 함수를 실행한다.
172      */
173     pthread_t col_thread;
174     pthread_create(&col_thread, NULL, check_columns, NULL);
175     /*
176      * 9개의 스레드를 생성하여 각 3x3 서브그리드를 검사하는 check_subgrid() 함수를 실행한다.
177      * 3x3 서브그리드의 위치를 식별할 수 있는 값을 함수의 인자로 넘긴다.
178      */
179     pthread_t subgrid_threads[9];
180     int arg[9];
181     for (i = 0; i < 9; ++i) {
182         arg[i] = i;
183     }
184
185     for (int k = 0; k < 9; ++k) {
186         pthread_create(subgrid_threads+k, NULL, check_subgrid, arg+k);
187     }
188     /*
189      * 11개의 스레드가 종료할 때까지 기다린다.
190      */
191     pthread_join(row_thread, NULL); // 행 검사 스레드
192     pthread_join(col_thread, NULL); // 열 검사 스레드
193     for (i = 0; i < 9; ++i) {       // 서브 그리드 검사 스레드
194         pthread_join(subgrid_threads[i], NULL);
195     }
```

`check_sudoku` 함수에서는 스도쿠의 행 검사, 열 검사, 서브 그리드 검사를 **다중 스레드**로 처리합니다. 이를 위해서는 각 업무를 맡을 스레드들을 생성하고, 모든 스레드가 검증을 완료할 때까지 대기해야합니다.

```
165     /*
166      * 스레드를 생성하여 각 행을 검사하는 check_rows() 함수를 실행한다.
167      */
168     pthread_t row_thread;
169     pthread_create(&row_thread, NULL, check_rows, NULL);
170     /*
171      * 스레드를 생성하여 각 열을 검사하는 check_columns() 함수를 실행한다.
172      */
173     pthread_t col_thread;
174     pthread_create(&col_thread, NULL, check_columns, NULL);
```

우선 행 검사 스레드와 열 검사 스레드를 새로 생성하여 각각의 업무(`check_rows` | `check_columns`)를 할당하였습니다. *참고로 11개의 스레드를 하나의 배열에 할당하지 않고, 업무에 따라 변수를 분리하여 스레드를 할당하였습니다.*

```
    /*
     * 9개의 스레드를 생성하여 각 3x3 서브그리드를 검사하는 check_subgrid() 함수를 실행한다.
     * 3x3 서브그리드의 위치를 식별할 수 있는 값을 함수의 인자로 넘긴다.
     */
    pthread_t subgrid_threads[9];
    int arg[9];
    for (i = 0; i < 9; ++i) {
        arg[i] = i;
    }

    for (int k = 0; k < 9; ++k) {
        pthread_create(subgrid_threads+k, NULL, check_subgrid, arg+k);
    }
```

다음으로는 9개의 서브 그리드를 검증할 스레드들을 추가하였습니다. 다만 앞서 행 검사 스레드와 열 검사 스레드와는 다르게, `check_subgrid` 함수는 *인자 값을 전달*받기 때문에 이를 유의해야 합니다.

특히 `pthread_create`을 통해 스레드를 생성하는 경우에는 함수의 인자 값을 **포인터** 형태로 전달해야 합니다. 이때 인자 값을 단 하나의 공간(= 변수)에 저장하여 전달하면, 하나의 저장 공간이 모든 스레드에 공유된 채로 진행되기 때문에 제대로 된 결과 값을 얻을 수 없을 것입니다. 따라서 각 스레드에 인자 값으로 넘겨줄 서브 그리드 번호를 서로 독립된 공간에 따로 저장해야 합니다.

이를 위해 우선 각각의 스레드에 인자로 전달할 서브 그리드 번호를 `arg`라는 배열에 저장하였습니다. 이후 스레드를 하나씩 생성하면서, 해당 스레드가 검사해야할 서브 그리드 번호가 저장된 공간의 주소를 전달했습니다. 이렇게 구현하게 되면 각각의 인자 값이 서로 다른 공간에 저장되어 있기 때문에 앞서 우려했던 상황을 해결할 수 있습니다.

```
    /*
     * 11개의 스레드가 종료할 때까지 기다린다.
     */
    pthread_join(row_thread, NULL); // 행 검사 스레드
    pthread_join(col_thread, NULL); // 열 검사 스레드
    for (i = 0; i < 9; ++i) {       // 서브 그리드 검사 스레드
        pthread_join(subgrid_threads[i], NULL);
    }
```

마지막으로 행 검증, 열 검증, 서브 그리드 검증이 모두 완료된 후에야 검증 결과가 올바른지 판단할 수 있기 때문에, 각각의 스레드가 검증을 완료할 때까지 기다릴 수 있도록 위와 같이 join을 진행하였습니다.

결과물 설명

```
********** BASIC TEST **********
 6 3 9 8 4 1 2 7 5
 7 2 4 9 5 3 1 6 8
 1 8 5 7 2 6 3 9 4
```

```
 2 5 6 1 3 7 4 8 9
 4 9 1 5 8 2 6 3 7
 8 7 3 4 6 9 5 2 1
 5 4 2 3 9 8 7 1 6
 3 1 8 6 7 5 9 4 2
 9 6 7 2 1 4 8 5 3
---
ROWS: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
COLS: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
---
 2 3 9 8 4 1 2 7 5
 7 6 4 9 5 3 1 6 8
 1 8 5 7 2 6 3 9 4
 2 5 6 1 3 7 4 8 9
 4 9 1 5 8 4 6 3 7
 8 7 3 2 6 9 5 2 1
 5 4 2 3 9 8 7 1 6
 3 1 8 6 7 5 9 3 2
 9 6 7 2 1 4 8 5 4
---
ROWS: (0,NO)(1,NO)(2,YES)(3,YES)(4,NO)(5,NO)(6,YES)(7,NO)(8,NO)
COLS: (0,NO)(1,NO)(2,YES)(3,NO)(4,YES)(5,NO)(6,YES)(7,NO)(8,NO)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
---
********** RANDOM TEST **********
 6 3 9 8 4 1 2 7 5
 7 2 4 9 5 3 1 6 8
 1 8 5 7 2 6 3 9 4
 2 5 6 1 3 7 4 8 9
 4 9 1 5 8 2 6 3 7
 8 7 3 4 6 9 5 2 1
 5 4 2 3 9 8 7 1 6
 3 1 8 6 7 5 9 4 2
 9 6 7 2 1 4 8 5 3
---
ROWS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,YES)(8,NO)
COLS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
---
 1 7 5 7 3 2 3 2 4
 4 6 5 2 5 2 8 2 5
 7 9 1 4 9 3 1 3 6
 9 7 1 9 2 8 9 8 4
 1 2 3 1 2 7 2 8 6
 2 9 3 5 3 3 8 3 7
 3 8 5 1 3 4 8 1 2
 9 2 6 2 6 5 9 5 4
 7 4 9 7 8 4 6 3 4
---
ROWS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,YES)
COLS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,NO)(8,YES)
---
 8 2 8 8 7 6 3 7 2
 8 7 9 9 1 2 5 3 2
 4 9 5 3 4 1 4 2 1
 6 2 9 5 8 6 6 2 5
```

```
 7 2 9 4 1 3 2 4 7
 4 8 5 9 1 5 7 1 9
 9 8 3 3 1 7 6 8 2
 1 6 5 3 5 8 1 7 5
 2 4 6 8 3 2 8 3 5
---
ROWS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
COLS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
---
 4 2 8 3 8 5 6 3 5
 5 6 2 4 1 5 3 2 5
 1 2 4 3 8 9 8 7 1
 4 2 6 3 2 5 2 5 3
 5 9 7 6 9 1 9 6 8
 1 3 8 8 4 7 4 7 1
 5 6 1 7 6 3 8 3 9
 3 7 8 8 9 5 4 6 1
 4 9 2 4 2 1 2 7 5
---
ROWS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
COLS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
---
 1 1 6 5 6 2 3 4 7
 4 8 3 3 8 1 4 5 2
 8 2 1 1 7 4 9 8 6
 7 3 6 8 6 1 6 5 7
 4 2 8 6 3 5 8 1 7
 1 7 9 2 6 1 3 4 5
 7 1 2 9 6 5 5 8 4
 5 8 3 5 4 7 9 2 1
 2 6 4 7 5 3 5 6 1
---
ROWS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
COLS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
---
 5 2 7 1 9 6 3 1 4
 1 8 9 2 7 4 9 2 5
 3 6 4 5 8 3 7 8 6
 7 1 2 4 8 3 1 2 5
 5 8 4 1 9 2 6 7 4
 3 9 6 5 7 6 9 3 8
 6 7 5 6 1 2 2 8 5
 3 4 9 4 7 5 4 7 9
 1 2 8 3 9 8 6 1 3
---
ROWS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
COLS: (0,NO)(1,NO)(2,NO)(3,NO)(4,NO)(5,NO)(6,NO)(7,NO)(8,NO)
GRID: (0,YES)(1,YES)(2,YES)(3,YES)(4,YES)(5,YES)(6,YES)(7,YES)(8,YES)
---
```

실행 결과 화면은 위와 같습니다.

결과를 살펴보면 `BASIC TEST` 과정에서는 스도쿠 검증이 정상적으로 이뤄진 반면, `RANDOM TEST` 과정에서는 마지막 테스트를 제외한 나머지 5개의 테스트에서 모두 실제 스도쿠와 결과가 매칭되지 않는 것을 확인할 수 있습니다.

우선, `BASIC TEST` 과정에서는 각각의 스도쿠를 하나씩 따로 검사하는 것을 확인할 수 있습니다. 기본 스도쿠 검사가 완료되면 스도쿠의 배치를 변경하여 다시 검증을 진행하고 있기 때문에, **두 검증 과정은 각각의 스도쿠 배열에 따라 정상적으로 진행**되고 있습니다.

```
271     printf("********** BASIC TEST **********\n");
272     /*
273      * 기본 스도쿠 퍼즐을 출력하고 검증한다.
274      */
275     check_sudoku();
276     for (i = 0; i < 9; ++i)
277         if (valid[0][i] == false || valid[1][i] == false || valid[2][i] ==
            false) {
278             printf("ERROR: 스도쿠 검증오류!\n");
279             return 1;
280         }
281     /*
282      * 기본 퍼즐에서 세 개를 맞바꾸고 검증해본다.
283      */
284     tmp = sudoku[0][0]; sudoku[0][0] = sudoku[1][1]; sudoku[1][1] = tmp;
285     tmp = sudoku[5][3]; sudoku[5][3] = sudoku[4][5]; sudoku[4][5] = tmp;
286     tmp = sudoku[7][7]; sudoku[7][7] = sudoku[8][8]; sudoku[8][8] = tmp;
287     check_sudoku();
```

반면에 `RANDOM TEST` 과정을 살펴보면 *스도쿠의 배열을 무작위로 섞는 과정*에서 **스도쿠 검증 과정**을 5번 시행하는 것을 확인할 수 있습니다. 현재 코드에서는 **스도쿠 배열**이 전역변수로 선언되어 있기 때문에, **서브 스레드**들은 이 스도쿠 배열을 서로 공유하게 됩니다. 즉, 하나의 **스레드에서 스도쿠 배열을 변경하게 되면 다른 스레드에서도 영향**을 받게 됩니다.

아래 코드에서 `tid 스레드`에서는 스도쿠 배열을 무작위로 섞고 있고, 이와 동시에 `check_sudoku`에서는 서브 스레드들이 스도쿠 배열을 검증하고 있습니다. 따라서 *스도쿠 배열 검증 과정*에서 값이 고정되어야 할 스도쿠 배열이 `tid 스레드`에 의해 계속해서 변경됨에 따라 검증이 제대로 이루어지지 않고 있습니다. 그렇기 때문에 5개의 검증 결과가 실제 스도쿠 배열과 일치하지 않는 것을 확인할 수 있습니다.

```
312     printf("********** RANDOM TEST **********\n");
313     /*
314      * 기본 스도쿠 퍼즐로 다시 바꾼 다음, shuffle_sudoku 스레드를 생성하여 퍼즐을 섞는다.
315      */
316     tmp = sudoku[0][0]; sudoku[0][0] = sudoku[1][1]; sudoku[1][1] = tmp;
317     tmp = sudoku[5][3]; sudoku[5][3] = sudoku[4][5]; sudoku[4][5] = tmp;
318     tmp = sudoku[7][7]; sudoku[7][7] = sudoku[8][8]; sudoku[8][8] = tmp;
319     if (pthread_create(&tid, NULL, shuffle_sudoku, NULL) != 0) {
320         fprintf(stderr, "pthread_create error: shuffle_sudoku\n");
321         return -1;
322     }
323     /*
324      * 무작위로 섞는 중인 스도쿠 퍼즐을 5번 검증해본다.
325      * 섞는 중에 검증하는 것이므로 결과가 올바르지 않을 수 있다.
326      * 충분히 섞을 수 있는 시간을 주기 위해 다시 검증하기 전에 1 usec 쉰다.
327      */
328     req.tv_sec = 0;
329     req.tv_nsec = 1000;
330     for (i = 0; i < 5; ++i) {
331         check_sudoku();
332         nanosleep(&req, NULL);
333     }
```

마지막 검증 과정에서는 `tid 스레드`가 스도쿠를 무작위로 배치하는 과정을 끝낼 때까지 기다린 후, 스도쿠 검증을 수행하기 때문에 결과가 올바르게 출력되는 것을 확인할 수 있습니다.

```
334     /*
335      * shuffle_sudoku 스레드가 종료될 때까지 기다린다.
336      */
337     alive = 0;
338     pthread_join(tid, NULL);
339     /*
340      * shuffle_sudoku 스레드 종료 후 다시 한 번 스도쿠 퍼즐을 검증해본다.
341      * 섞는 과정이 끝났기 때문에 퍼즐 출력과 검증 결과가 일치해야 한다.
342      */
343     check_sudoku();
```

느낀점

이번 프로젝트를 진행하면서 다중 스레드를 직접 생성하고 이를 실행해보면서, 다중 스레드를 활용한 코드에서는 어떤 식으로 코드의 흐름이 진행되는지 알아볼 수 있었습니다. 그리고 작업량이 많은 작업을 여러 작업들로 쪼개고, 각각을 여러 스레드에게 할당하여 이를 병렬적으로 진행하는 과정을 실제 코드로 적용해볼 수 있었습니다. 예전에는 많은 작업량을 수행하는 코드를 단 하나의 스레드로만 처리를 했다면, 이번 과제를 계기로 많은 작업량이 포함된 프로젝트에서 다중 스레드 개념을 도입할 수 있을 것입니다.

특히 현재 이 프로젝트에서는 `RANDOM TEST` 과정에서 결과가 제대로 출력되지 않는 문제점이 있는데, 이 원인이 **스도쿠 배열**이 ***공유 자원***으로 사용되고 있다는 점과 이를 해결하기 위해서는 공유 자원 접근에 대한 제어가 필요하다는 것을 예측할 수 있었습니다. 특히 다중 스레드로 코드를 처리할 때는 공유 자원의 특성에 의해 원하는 결과를 얻지 못할 상황에 대비하여 이를 적절하게 관리하는 것이 중요하다는 사실을 깨달을 수 있었습니다.

그리고 `check_sudoku`에서 check_subgrid 스레드를 여러 개 생성할 때, 인자 값을 하나의 변수에 할당하고 모든 스레드에 해당 변수의 포인터를 전달하는 실수를 범하기도 하였습니다.

이 실수를 해결하는 과정에서는 포인터 개념에 대해서 다시금 상기하며 복습할 수 있었습니다.