

北京交通大学操作系统实验报告

姓名：程维森

学号：21231264

一. 实验内容

实验标题：Multithreaded programming

内容：

1. Sudoku Solution Validator
2. Multithreaded Sorting Application

内容简要概括:

- 1.这个项目要求设计一个多线程应用程序，用于验证数独 (Sudoku) 谜题的解是否合法。数独谜题是一个 9×9 的网格，每一列、每一行和每个 3×3 的子网格必须包含数字 1 到 9。为了实现这个目标，需要创建多个线程，分别检查每一列、每一行和每个 3×3 子网格是否包含了 1 到 9 的所有数字。
- 2.编写一个多线程排序程序，工作原理如下：将一组整数分成两个相等大小的子列表。然后使用两个单独的线程（称为排序线程）分别对这两个子列表进行排序，排序算法可以自行选择。接着，由第三个线程——合并线程 (merging thread) 将这两个子列表合并成一个已排序的单一列表。

二. 实验一

1. 实验思路分析:

我们可以将整个项目的思路分析为以下步骤:

1. 定义数据结构和初始化:

定义一个 9×9 的二维数组 `shudu` 作为数独的初始布局。

创建一个标记数组 `line` 用于记录数字是否已经出现。

创建一个二维数组 `check` 用于标记检查结果，0 表示合法，1 表示不合法。

2. 创建多线程:

根据给定的策略，创建多个线程来并行检查数独的不同部分。

创建九个列检查线程，每个线程检查数独的一列，确保包含了 1 到 9 的所有数字。

创建九个行检查线程，每个线程检查数独的一行，确保包含了 1 到 9 的所有数字。

创建九个子网格检查线程，每个线程检查数独的一个 3×3 子网格，确保包含了 1 到 9 的所有数字。

3. 多线程处理逻辑:

每个线程在开始时初始化 `line` 数组，然后遍历特定的区域（行、列、子网格）。

对于每个数字，检查它是否已经在当前区域中出现。

如果数字没有出现过，将 `line` 数组对应位置标记为 1。

如果数字已经出现过，将相应的 `check` 数组位置标记为 1，表示该区域不合法。

4. 结果判断:

所有线程执行完毕后，检查 `check` 数组的值来确定数独解是否合法。

如果 `check` 数组中所有元素都为 0，表示数独解合法，输出 "legal"。

如果 `check` 数组中存在 1，表示数独解不合法，输出 "illegal"。

2. 实验函数解析:

1. `checkx` 函数：

1. 初始化操作：将`line`数组中的所有元素初始化为 0，`line`数组用于标记数字是否已经出现过。
2. 获取参数：从传入的参数中获取行数 `x`。
3. 检查列合法性：遍历该行的所有列，对于每个数字，检查它是否已经在该行中出现。
 - 如果没有出现过，将`line`数组对应位置标记为 1。
 - 如果已经出现过，将`check[0][x]`标记为 1，表示该列不合法。
- 4.返回结果：返回 NULL，因为该函数没有需要返回的结果。

```
void *checkx(void *arg){
    for(int i = 0; i <= 100; i++)
        line[i] = 0;
    int x = *(int *)arg;
    for (int i = 0; i < 9; i++) {
        if (!line[shudu[x][i]]) {
            line[shudu[x][i]] = 1;
        } else {
            check[0][x] = 1;
        }
    }

    return NULL;
}
```

2. `checky` 函数：

1. 初始化操作：同样将`line`数组中的所有元素初始化为 0。
2. 获取参数：从传入的参数中获取列数 `y`。
3. 检查行合法性：遍历该列的所有行，对于每个数字，检查它是否已经在该列中出现。
 - 如果没有出现过，将`line`数组对应位置标记为 1。
 - 如果已经出现过，将`check[1][y]`标记为 1，表示该行不合法。
- 4.返回结果：返回 NULL，因为该函数没有需要返回的结果。

```

void *checky(void *arg){
    for(int i = 0; i <= 100; i++)
        line[i] = 0;
    int y = *(int *)arg;
    for (int i = 0; i < 9; i++) {
        if (!line[shudu[i][y]]) {
            line[shudu[i][y]] = 1;
        } else {
            check[1][y] = 1;
        }
    }

    return NULL;
}

```

3. 'checkangel' 函数：

1. 初始化操作：同样将`line`数组中的所有元素初始化为 0。
2. 获取参数：从传入的参数中获取一个`node`结构体，其中包含了一个 3x3 子网格的左上角坐标 `(px, py)`。
3. 检查子网格合法性：遍历该子网格的所有元素，对于每个数字，检查它是否已经在该子网格中出现。
 - 如果没有出现过，将`line`数组对应位置标记为 1。
 - 如果已经出现过，将`check[2][px / 3 * 3 + py / 3]`标记为 1，表示该子网格不合法。这里的计算用于确定在`check`数组中的位置。
4. 返回结果：返回 NULL，因为该函数没有需要返回的结果。

```

void *checkangel(void *arg){
    for(int i = 0;i <= 100;i++){
        line[i] = 0;
    }
    node p = *(node *)arg;
    int px = p.x,py = p.y;
    for(int i = px;i <= px + 2;i++){
        for(int j = py;j <= py + 2;j++){
            if (!line[shudu[i][j]]){
                line[shudu[i][j]] = 1;
            }
            else{
                check[2][px / 3 * 3 + py / 3] = 1;
            }
        }
    }
    return NULL;
}

```

这三个函数的设计思路都是类似的：初始化一个标记数组，用来记录数字的出现情况，然后遍历特定的区域（行、列、子网格），检查数字是否已经出现，如果出现则标记相应的位置为 1 表示不合法。整体上，这个设计采用了多线程的方式并行地检查数独的不同部分，提高了程序的运行效率。

3. 多线程处理函数逻辑:

```

int main(){
    int flag = 1;
    pthread_t s[10][10];
    for (int i = 0; i < 9; i++) {
        pthread_create(&s[0][i], NULL, checkx, &i);
        pthread_join(s[0][i], NULL);
    }
    for (int i = 0; i < 9; i++) {
        pthread_create(&s[1][i], NULL, checky, &i);
        pthread_join(s[1][i], NULL);
    }
    for (int i = 0; i < 9; i += 3) {
        for (int j = 0; j < 9; j += 3) {
            node tmp = {i, j};
            pthread_create(&s[2][i + j / 3], NULL, checkangel, &tmp);
            pthread_join(s[2][i + j / 3], NULL);
        }
    }

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 9; j++) {
            if (check[i][j]) {
                flag = 0;
                break;
            }
        }
    }

    puts("If the thread is legal, it will output legal; if it is illegal, it will output illegal.");
    if(flag){
        printf("legal\n");
    }
    else{
        printf("illegal\n");
    }
    return 0;
}

```

这个主函数的逻辑是为了检查给定的数独（Sudoku）是否合法。下面是逐步解释这段代码的逻辑：

1. `int flag = 1;`：初始化一个标志位 `flag` 为 1，表示数独合法。
2. `pthread_t s[10][10];`：定义一个二维数组 `s`，用于存储线程标识符。
3. 检查每一列的合法性：
 - 通过循环创建 9 个线程，每个线程调用 `checkx` 函数来检查一个特定的列。线程的编号 `i` 从 0 到 8。
 - 每个线程的参数是 `&i`，即当前列的索引。
 - 使用 `pthread_join` 函数等待每个线程执行完毕，然后再继续下一个循环。
4. 检查每一行的合法性：
 - 通过循环创建 9 个线程，每个线程调用 `checky` 函数来检查一个特定的行。线程的编号 `i` 从 0 到 8。
 - 每个线程的参数是 `&i`，即当前行的索引。
 - 使用 `pthread_join` 函数等待每个线程执行完毕，然后再继续下一个循环。
5. 检查每个子网格的合法性：
 - 通过两层循环，外层循环从 0 到 6，内层循环从 0 到 6。
 - 创建 9 个线程，每个线程调用 `checkangel` 函数来检查一个 3x3 子网格的合法性。
 - 每个线程的参数是一个 `node` 结构体，表示子网格的左上角坐标。
 - 使用 `pthread_join` 函数等待每个线程执行完毕，然后再继续下一个循环。

6. 检查结果:

使用两层循环遍历 `check` 数组,如果发现任何一个子网格不合法(`check[i][j] == 1`),将 `flag` 设置为 0,表示数独解不合法。

最终,通过 `flag` 的值,程序判断数独解是否合法。如果 `flag` 为 1,表示数独解合法,输出 "legal"。如果 `flag` 为 0,表示数独解不合法,输出 "illegal"。但是,需要注意之前提到的问题: `pthread_join` 函数的位置导致线程无法并行执行,需要将其移到所有线程创建完毕之后的位置,以实现并行处理。

4. 总结:

初始化和数据结构定义:

初始化数独布局和辅助数据结构,包括 shudu 数组、line 数组和 check 数组。

多线程任务划分:

设计了三个不同的线程函数 checkx、checky 和 checkangel,分别用于检查数独的列、行和子网格的合法性。

使用多个线程同时处理不同部分,提高了程序的运行效率。

线程创建和等待问题:

项目中的线程创建和等待逻辑存在问题。线程在创建后立即被等待,导致无法并行处理。pthread_join 函数应该放在所有线程创建完毕后的位置,以实现并行执行。

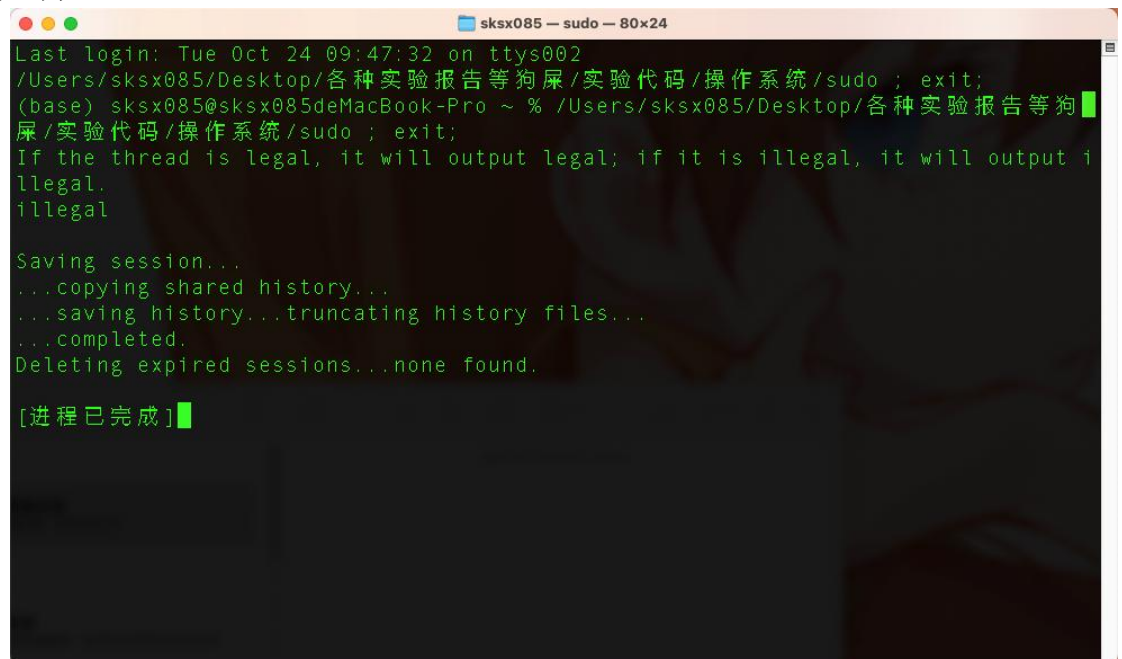
并行处理和结果判断:

通过并行处理,程序同时检查了数独的所有列、行和子网格的合法性。检查结果存储在 check 数组中,通过检查 check 数组的值,判断数独解是否合法。

输出结果:

根据 flag 的值(合法性标志位),输出 "legal" 或 "illegal",表示数独解是否符合规则。

运行示例:



```
sksx085 — sudo — 80x24
Last login: Tue Oct 24 09:47:32 on ttys002
/Users/sksx085/Desktop/各种实验报告等狗屎/实验代码/操作系统/sudo ; exit;
(base) sksx085@sksx085deMacBook-Pro ~ % /Users/sksx085/Desktop/各种实验报告等狗屎/实验代码/操作系统/sudo ; exit;
If the thread is legal, it will output legal; if it is illegal, it will output illegal.
illegal

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.
Deleting expired sessions...none found.

[进程已完成]
```

三. 实验二

1. 实验思路分析:

1.输入数据和分割子列表：首先，从用户输入中获取一组整数。然后，将这组整数分成两个相等大小的子列表。在代码中，你可以使用一个数组（例如 `ori` 数组）来存储这组整数。

2.创建排序线程：接下来，创建两个排序线程。每个排序线程负责对一个子列表进行排序。在代码中，使用 `pthread_create` 函数创建这两个排序线程。你可以定义一个结构体（例如 `datas` 结构体）来传递排序的起始和结束索引给排序线程。

3.等待排序线程完成：使用 `pthread_join` 函数等待两个排序线程完成排序。这确保了排序线程执行完毕，可以安全地进行合并操作。

4.创建合并线程：创建一个合并线程，该线程负责将两个已排序的子列表合并成一个单一的已排序列表。在代码中，使用 `pthread_create` 函数创建合并线程，并传递需要合并的子列表的起始索引和结束索引。

5.等待合并线程完成：使用 `pthread_join` 函数等待合并线程完成。这确保了合并线程执行完毕，可以安全地输出最终的排序结果。

6.输出排序结果：在合并线程完成后，输出合并后的已排序列表。在代码中，你需要分配内存来存储排序结果，并且在最后释放这块内存。

2.实验函数解析

1.quick_sort(int start, int end)

这是一个递归实现的快速排序算法。以下是函数的解析：

1.参数：

`start`：当前子数组的起始索引。

`end`：当前子数组的结束索引。

2.基准条件：

如果 `start >= end`，表示当前子数组只包含一个元素或者没有元素，无需排序，直接返回。

3.选择基准元素：

选择 `ori[start]` 作为基准元素（pivot）。

4.分区过程：

使用两个指针 `i` 和 `j` 分别从子数组的开头和结尾向中间移动，直到找到元素不满足排序条件（`ori[j] < key` 或 `ori[i] > key`）。

交换 `ori[i]` 和 `ori[j]`，将小于基准元素的元素放到左侧，大于基准元素的元素放到右侧。

5.递归调用：

递归调用 `quick_sort(start, i - 1)` 对左侧子数组进行排序。

递归调用 `quick_sort(i + 1, end)` 对右侧子数组进行排序。

6.特点：

快速排序是一种分治算法，具有较好的平均和最坏情况下的性能。它通过不断地将问题分解为更小的子问题，然后解决这些子问题。


```

void quick_sort(int start, int end) {
    if (start >= end)
        return;
    int key = ori[start];
    int i = start, j = end;
    while (i < j) {
        while (i < j && ori[j] >= key) {
            j--;
        }
        ori[i] = ori[j];
        while (i < j && ori[i] <= key) {
            i++;
        }
        ori[j] = ori[i];
    }
    ori[i] = key;
    quick_sort(start, i - 1);
    quick_sort(i + 1, end);
}

```

2.merge(int *ori, int *sorted_num, int len, int div)

这个函数负责将两个已排序的子数组合并成一个单一的已排序列表。以下是函数的解析：

1.参数：

- ori：原始整数数组。
- sorted_num：用于存储合并结果的数组。
- len：子数组的长度。
- div：两个子数组的分割点索引。

2.初始化指针：

使用指针 i 和 j 分别指向两个子数组的起始位置，使用指针 p 和 q 分别指向两个子数组的中间位置。

3.合并过程：

在合并的过程中，比较两个子数组的元素，将较小的元素放入 sorted_num 数组中，然后将对应指针向后移动。

4.处理剩余元素：

如果其中一个子数组的元素已全部放入 sorted_num 中，就将另一个子数组中的剩余元素依次放入 sorted_num。

5.特点：

合并排序的时间复杂度较低，但它需要额外的空间来存储临时结果（sorted_num 数组）。该函数使用两个指针来遍历两个子数组，按顺序将较小的元素放入 sorted_num 中，直到两个子数组都被遍历完毕。

```

void merge(int *ori, int *sorted_num, int len, int div) {
    int i = 0, j = div, p = div + 1, q = len - 1;
    int m = 0;
    while (i <= j && p <= q) {
        if (ori[i] <= ori[p]) {
            sorted_num[m++] = ori[i++];
        } else {
            sorted_num[m++] = ori[p++];
        }
    }
    while (i <= j) {
        sorted_num[m++] = ori[i++];
    }
    while (p <= q) {
        sorted_num[m++] = ori[p++];
    }
}

```

以上两个函数共同实现了快速排序和归并排序的功能，分别负责排序两个子数组并将它们合并成一个有序的结果数组。

3.多线程处理逻辑：

这段代码实现了多线程的排序逻辑，下面是对代码的逐步解释：

1.用户输入：

- 用户输入一组整数，并且这些整数被存储在 `ori` 数组中。

2.分割子数组：

- 计算 `div`，将 `ori` 数组分成两个相等大小的子数组。`div` 是第一个子数组的最后一个索引。

3. 创建排序线程：

- 创建了两个排序线程，每个线程负责一个子数组的排序。
- 使用 `pthread_create` 函数创建线程，传递给线程的参数是包含子数组起始和结束索引的 `datas` 结构体。

4. 等待排序线程结束：

- 使用 `pthread_join` 函数等待两个排序线程完成。

5. 动态内存分配：

- 使用 `malloc` 分配了一个足够存储排序结果的内存空间。如果内存分配失败，会打印错误信息并退出程序。

6. 归并排序结果：

- 调用 `merge` 函数，将两个已排序的子数组合并成一个有序的数组，结果存储在 `sorted_num` 中。

7. 输出排序后的数组：

- 打印排序后的数组。

8. 释放内存：

- 使用 `free` 函数释放动态分配的内存。

多线程的逻辑体现在步骤 3 和 4 中。在步骤 3 中，通过创建两个排序线程，实现了对两个子数组的并行排序。在步骤 4 中，使用 `pthread_join` 函数等待这两个排序线程的完成，确保在进行合并操作之前，两个子数组都已经排序完成。这样，在步骤 6 中的合并操作就可以安全地进行了。整个过程中，多线程的并行执行提高了排序的效率。

```
int main() {
    printf("Input ori:\n");
    int len = 0;
    char c;
    while ((c = getchar()) != '\n') {
        ungetc(c, stdin);
        scanf("%d", &ori[len++]);
    }

    int div = len / 2 - 1;

    pthread_t threads[MAX_THREADS];
    datas s1_par = {0, div};
    datas s2_par = {div + 1, len - 1};

    // 创建快速排序线程
    pthread_create(&threads[0], NULL, thread_sort, &s1_par);
    pthread_create(&threads[1], NULL, thread_sort, &s2_par);

    // 等待快速排序线程结束
    for (int i = 0; i < 2; ++i) {
        pthread_join(threads[i], NULL);
    }

    // 归并两个已排序的子数组
    int *sorted_num = (int *)malloc(len * sizeof(int));
    if (sorted_num == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE); // 退出程序，表示内存分配失败
    }

    merge(ori, sorted_num, len, div);

    // 输出排序后的数组
    printf("Sorted array:\n");
    for (int i = 0; i < len; ++i) {
        printf("%d ", sorted_num[i]);
    }
    printf("\n");

    // 释放动态分配的内存
    free(sorted_num);

    return 0;
}
```

4.总结:

在上述代码中,多线程处理用于加速排序操作,主要涉及到两个重要的多线程排序算法:快速排序(`quick_sort`)和归并排序(`merge`)。

1. 快速排序(`quick_sort`) :

-功能:快速排序是一种分治算法,它将一个数组分成两个子数组,然后递归地对子数组进行排序,最终将整个数组排序。

-多线程处理:在代码中,通过创建两个排序线程,每个线程负责一个子数组的排序。通过多线程的方式,实现了并行处理,加速了排序过程。

-算法特点:快速排序的时间复杂度为 $O(n \log n)$,在平均情况下具有较好的性能。

2. 归并排序(`merge`) :

-功能:归并排序是一种稳定的、比较排序算法,它将两个已排序的子数组合并成一个有序的数组。

-多线程处理:在代码中,使用了一个合并线程,负责将两个已排序的子数组并行地合并为一个有序数组。这种并行处理提高了归并的效率。

-算法特点:归并排序的时间复杂度同样为 $O(n \log n)$,具有稳定性,适用于大规模数据的排序。

总结:

1. 分治策略:快速排序和归并排序都是基于分治思想的排序算法,通过递归地分割问题并分别解决问题,最后将结果合并,实现整体排序。

2. 多线程优势:多线程处理加速了排序操作,通过并行处理,充分利用了多核心处理器的性能,提高了排序的效率。

3. 内存管理:在多线程排序过程中,需要注意内存的分配和释放,确保线程安全,同时避免内存泄漏。

4. 稳定性:归并排序具有稳定性,相等元素的相对顺序不会发生改变,适用于对稳定性有要求的场景。

5. 并行性控制:在多线程处理中,需要确保线程的同步,比如使用`pthread_join`函数等待线程结束,保证合并线程在两个子数组排序完成后再进行操作。

综上所述,以上的多线程处理方式展示了如何将并行计算应用于排序算法,充分发挥多核心处理器的性能,加速了排序的过程。同时,这种多线程的排序实现也为其他排序算法的并行化提供了思路和参考。

运行示例:

```
sksx085 — sort — 80x24
Last login: Wed Oct 25 09:15:14 on ttys004
/Users/sksx085/Desktop/各种实验报告等狗屎/实验代码/操作系统/sort ; exit;
(base) sksx085@sksx085deMacBook-Pro ~ % /Users/sksx085/Desktop/各种实验报告等狗屎/实验代码/操作系统/sort ; exit;
Input ori:
1 6 8 9 0 3 8 5 100
Sorted array:
0 1 3 5 6 8 8 9 100

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

四. 实验代码

请参考: <https://github.com/sksx085/OSHomework>