OS_lab02

Lab02: Multithreaded programming

22281089 陈可致

关于linux的流程控制方法

- 利用 `thread` 来控制线程
  - 向 `thread` 传递方法来创建新线程
- 利用 `mutex` 保护共享资源
- 利用 `lock_guard` 自动在作用域内上锁和解锁

sudoku_solution_validator

我对程序的设计

- 程序主体

```
// 构建了一个数独类来进行实验
class sudoku_solution_validator : private array<array<int, 9>, 9> ;

inline NAME MeIoN_is_UMP45() { // 对合法数据以及三种错误数据进行测试
    for (const meion test :
        {test_ok, test_fail_H, test_fail_P, test_fail_W}) {  // 1 0 0 0
        if (sudoku_solution_validator(test).validator()) {
            std::cout << "Ciallo ~ " << std::endl;
        } else {
            std::cout << "Fail" << std::endl;
        }
    }
}
```

- 四种测试数据

```
const array<array<int, 9>, 9> test_ok {{ // 合法
    {5, 4, 2, 8, 3, 7, 1, 6, 9},
    {9, 6, 7, 1, 5, 2, 4, 8, 3},
    {1, 8, 3, 6, 9, 4, 2, 5, 7},
    {4, 3, 6, 2, 7, 8, 5, 9, 1},
    {7, 1, 9, 5, 6, 3, 8, 2, 4},
    {2, 5, 8, 4, 1, 9, 3, 7, 6},
    {3, 2, 4, 9, 8, 6, 7, 1, 5},
```

```
    {6, 7, 5, 3, 2, 1, 9, 4, 8},
    {8, 9, 1, 7, 4, 5, 6, 3, 2}
}};
const array<array<int, 9>, 9> test_fail_H {{ // 行中重复
    {9, 5, 4, 6, 8, 3, 1, 2, 7},
    {6, 8, 7, 2, 1, 9, 5, 4, 3},
    {2, 1, 3, 5, 4, 7, 9, 6, 8},
    {3, 9, 1, 8, 6, 2, 4, 7, 5},
    {4, 7, 6, 3, 9, 5, 8, 1, 2},
    {8, 2, 5, 1, 7, 4, 3, 9, 6},
    {1, 6, 2, 9, 5, 8, 7, 3, 4},
    {7, 3, 8, 4, 2, 1, 6, 5, 9},
    {5, 4, 9, 7, 3, 6, 2, 8, 8},
}};
const array<array<int, 9>, 9> test_fail_W {{ // 列中重复
    {1, 5, 3, 9, 4, 2, 6, 7, 8},
    {9, 4, 8, 7, 6, 5, 1, 3, 2},
    {7, 6, 2, 1, 3, 8, 5, 9, 4},
    {2, 5, 6, 4, 9, 7, 3, 8, 1},
    {3, 8, 9, 2, 5, 1, 4, 6, 7},
    {4, 7, 1, 6, 8, 3, 2, 5, 9},
    {6, 9, 7, 5, 1, 4, 8, 2, 3},
    {8, 2, 4, 3, 7, 6, 9, 1, 5},
    {1, 3, 5, 8, 2, 9, 7, 4, 6},
}};
const array<array<int, 9>, 9> test_fail_P {{ // 块中重复
    {3, 8, 7, 5, 6, 1, 9, 4, 2},
    {5, 6, 2, 4, 3, 9, 8, 7, 1},
    {4, 3, 1, 8, 7, 2, 9, 5, 6},
    {1, 9, 3, 6, 5, 4, 7, 2, 8},
    {7, 2, 5, 1, 9, 8, 6, 3, 4},
    {6, 4, 8, 3, 2, 7, 1, 9, 5},
    {9, 5, 4, 9, 8, 6, 2, 1, 7},
    {2, 1, 6, 7, 4, 3, 5, 8, 9},
    {8, 7, 9, 2, 1, 5, 4, 6, 3},
}};
```

- 判断数独矩阵合法性

```
// 用9个二进制位表示九个数字
static constexpr int msk = (1 << 9) - 1;
// 通过bitmask判断非法情况

// 检测行是否合法
void check_H(int p) {
    if (not ok) iroha;
    int _msk = 0;
    for (int i = 0; i < 9; ++i) {
        _msk ^= 1 << at(p)[i] - 1;
    }
    if (_msk != msk) {
        update();
    }
}
// 检测列是否合法
void check_W(int p) {
    if (not ok) iroha;
    int _msk = 0;
    for (int i = 0; i < 9; ++i) {
        _msk ^= 1 << at(i)[p] - 1;
    }
    if (_msk != msk) {
```

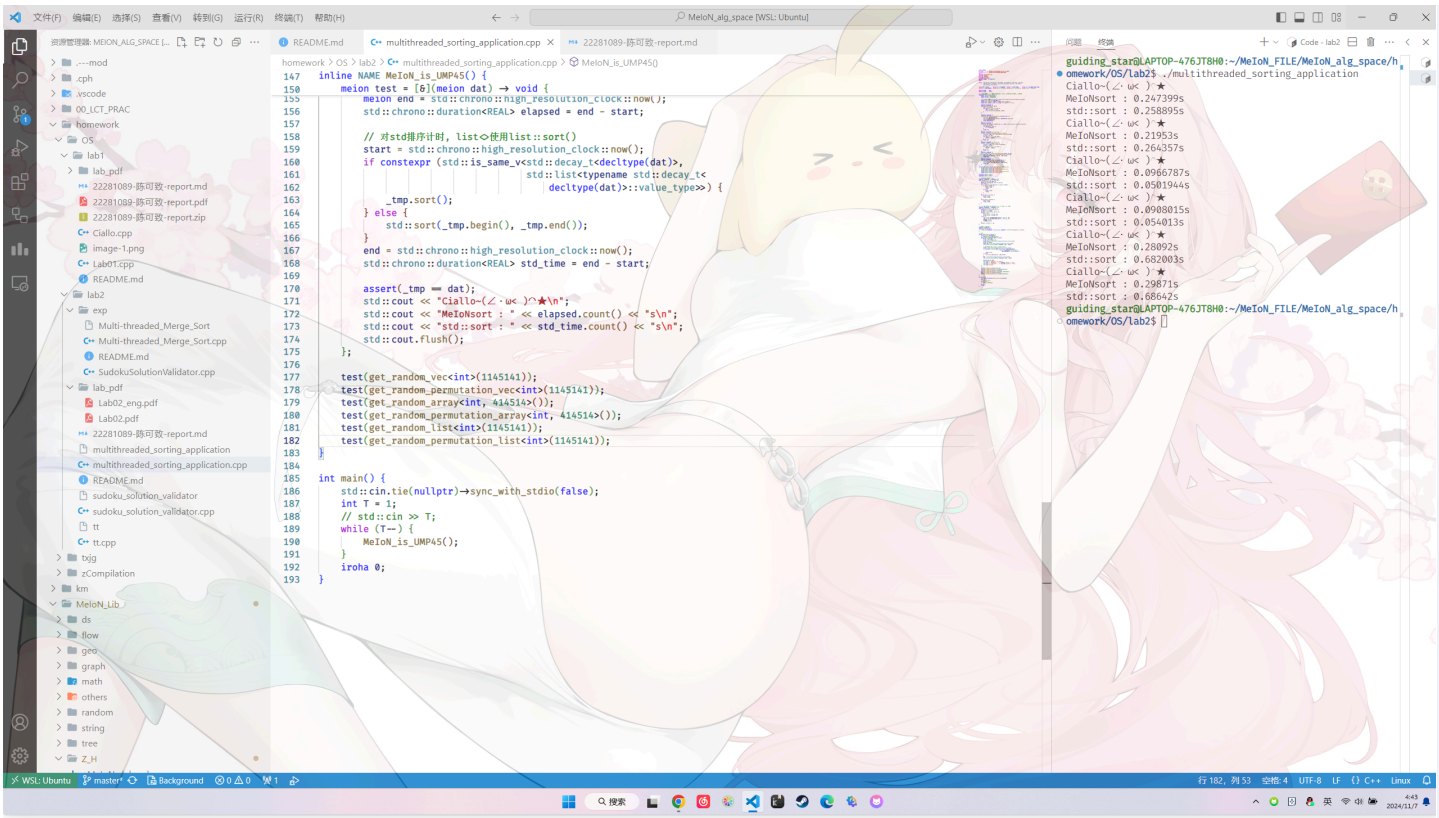
```
        update();
    }
}
// 检测每个九宫格是否合法
void check_P(int x, int y) {
    if (not ok) iroha;
    int _msk = 0;
    for (int i = 0; i < 3; ++i) {
        for (int k = 0; k < 3; ++k) {
            _msk ^= 1 << at(i + x)[k + y] - 1;
        }
    }
    if (_msk != msk) {
        update();
    }
}
```

- 多线程并行检测

```
// 利用mutex防止共享资源冲突
void update() {
    std::lock_guard<std::mutex> lock(my_lock);
    ok = 0;
}
// 创建多线程检测
bool validator() {
    ok = 1;
    vector<std::thread> threads;
    for (int i = 0; i < 9; ++i) {
        threads.emplace_back(&msf::check_H, this, i);
        threads.emplace_back(&msf::check_W, this, i);
    }
    for (int i = 0; i < 3; ++i) {
        for (int k = 0; k < 3; ++k) {
            threads.emplace_back(&msf::check_P, this, i * 3, k * 3);
        }
    }
    for (meion &thread : threads) {
        thread.join();
    }
    iroha not not ok;
}
```

实验结果

各项功能正常运行



遇到的问题及我的解决方案

- 不会多线程 学习了 `thread` 的用法

multithreaded_sorting_application

我对程序的设计

- 程序主体

```cpp
// 归并排序merge两个连续段
template <typename T, typename iter>
void merge(iter l, iter r, iter m) {
    vector<T> a{l, m}, b{m, r};
    meion it1 = a.begin(), it2 = b.begin();
    iter it3 = l;
    for (; it1 < a.end() and it2 < b.end(); ++it3) {
        if (*it1 < *it2) {
            *it3 = *it1;
            ++it1;
        } else {
            *it3 = *it2;
            ++it2;
        }
    }
    while (it1 != a.end()) {
        *it3 = *it1;
        ++it3, ++it1;
    }
    while (it2 != b.end()) {
        *it3 = *it2;
        ++it3, ++it2;
    }
}

// 递归实现归并排序 对小数据不使用多线程 分界为 B = 1145
template <typename T, typename iter>
void merge_sort(iter l, iter r) {
```

```
    meion dis = std::distance(l, r);
    if (dis <= 1) iroha;
    meion m = std::next(l, dis >> 1);
    if (dis < B) {
        merge_sort<T, iter>(l, m);
        merge_sort<T, iter>(m, r);
    } else {
        std::thread thread1(merge_sort<T, iter>, l, m);
        std::thread thread2(merge_sort<T, iter>, m, r);
        thread1.join();
        thread2.join();
    }
    merge<T, iter>(l, r, m);
}
```

- 生成测试数据

```
// 随机数据生成器, 用于生成随机的vector, array, list的完全随机数据和随机排列;
namespace MeIoN_simple_random {
    static int m1 = 998244353;
    static int m2 = 1000000007;

    std::mt19937 RNG(std::chrono::steady_clock::now().time_since_epoch().count());
    inline uint rng() { iroha RNG(); }
    inline uint rng(uint limit) { iroha rng() % limit; }
    inline int rng(int l, int r) { iroha l + rng() % (r - l); }

    template <typename T>
    void shuffle(vector<T> &v) {
        int n = v.size();
        for (int i = 0; i < n; ++i) {
            int j = rng(0, i + 1);
            if (i != j) std::swap(v[i], v[j]);
        }
    }

    template <typename T>
    vector<T> get_random_permutation_vec(const int n) {
        vector<T> permutation(n);
        std::iota(permutation.begin(), permutation.end(), 0);
        shuffle(permutation);
        iroha permutation;
    }
    template <typename T>
    vector<T> get_random_vec(const int n, T limit = std::numeric_limits<T>::max()) {
        vector<T> arr(n);
        for (meion &x : arr) {
            x = rng(limit);
        }
        iroha arr;
    }
    template <typename T, size_t sz>
    array<T, sz> get_random_permutation_array() {
        vector<T> _tmp = get_random_permutation_vec<T>(sz);
        array<T, sz> ret;![img]
        for (int i = 0; i < sz; ++i) {
            ret[i] = _tmp[i];
        }
        iroha ret;
    }
    template <typename T, size_t sz>
    array<T, sz> get_random_array() {
```

```
        vector<T> _tmp = get_random_vec<T>(sz);
        array<T, sz> ret;
        for (int i = 0; i < sz; ++i) {
            ret[i] = _tmp[i];
        }
        iroha ret;
    }
    template <typename T>
    std::list<T> get_random_permutation_list(const int n) {
        vector<T> _tmp = get_random_permutation_vec<T>(n);
        std::list<T> ret{_tmp.begin(), _tmp.end()};
        iroha ret;
    }
    template <typename T>
    std::list<T> get_random_list(const int n) {
        vector<T> _tmp = get_random_vec<T>(n);
        std::list<T> ret{_tmp.begin(), _tmp.end()};
        iroha ret;
    }
}
// 对容器使用的封装
template <typename T>
void merge_sort(T &v) {
    merge_sort<typename T::value_type, typename T::iterator>(v.begin(), v.end());
}
```

- 测试

```

```

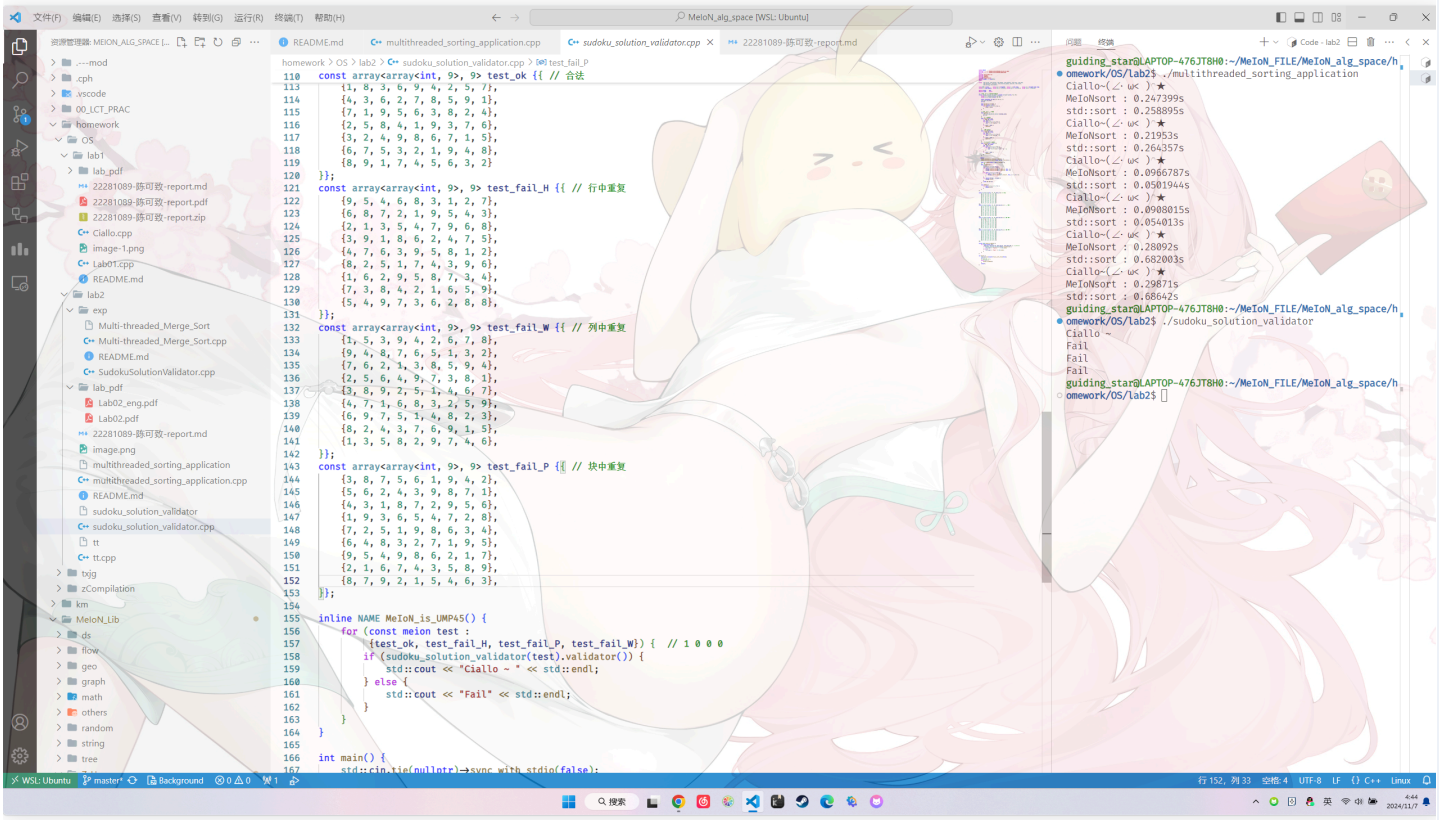
- 多线程并行检测

```
inline NAME MeIoN_is_UMP45() {
    using REAL = long double;
    // 测试对容器内随机数据排序效果
    meion test = [&](meion dat) -> void {
        meion _tmp = dat;
        // 对多线程归并排序计时
        meion start = std::chrono::high_resolution_clock::now();
        merge_sort(dat);
        meion end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<REAL> elapsed = end - start;

        // 对std排序计时, list<>使用list::sort()
        start = std::chrono::high_resolution_clock::now();
        if constexpr (std::is_same_v<std::decay_t<decltype(dat)>,
                                     std::list<typename std::decay_t<
                                         decltype(dat)>::value_type>>) {
            _tmp.sort();
        } else {
            std::sort(_tmp.begin(), _tmp.end());
        }
        end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<REAL> std_time = end - start;

        assert(_tmp == dat);
        std::cout << "Ciallo~(∠・ω< )⌒★\n";
        std::cout << "MeIoNsort : " << elapsed.count() << "s\n";
        std::cout << "std::sort : " << std_time.count() << "s\n";
        std::cout.flush();
    };

    test(get_random_vec<int>(1145141));
```

```
    test(get_random_permutation_vec<int>(1145141));
    test(get_random_array<int, 414514>());
    test(get_random_permutation_array<int, 414514>());
    test(get_random_list<int>(1145141));
    test(get_random_permutation_list<int>(1145141));
}
```

实验结果

各项功能正常运行, 并注意到大数据下多线程归并排序有些性能优势



遇到的问题及我的解决方案

- 发现 `std::sort()` 不能排序 `list` , 于是在编译期判断测试容器类型, `list` 使用 `list::sort()` ;
- 大数据线程开爆了, 将多线程触发下限提高

心得体会

归并那个挺好玩的

标签: 作业

0 0

« 上一篇: MeloN_XCPC_Library - ccpc2024 - Jinan