操作系统实验 高级进程间通信 快速排序问题

姓名：陈彦旭

班级：无24

1 实验目的

1. 通过对进程间高级通信问题的编程实现，加深理解进程间高级通信的原理；
2. 对 Windows 或 Linux 涉及的几种高级进程间通信机制有更进一步的了解；
3. 熟悉 Windows 或 Linux 中定义的与高级进程间通信有关的函数。

2 问题描述

对于有 1,000,000 个乱序数据的数据文件执行快速排序。

3 实验步骤

（1）首先产生包含 1,000,000 个随机数（数据类型可选整型或者浮点型）的数据文件；

（2）每次数据分割后产生两个新的进程（或线程）处理分割后的数据，每个进程（线程）处理的数据小于 1000 以后不再分割（控制产生的进程在 20 个左右）；

（3）线程（或进程）之间的通信可以选择下述机制之一进行：管道（无名管道或命名管道），消息队列，共享内存。

（4）通过适当的函数调用创建上述 IPC 对象，通过调用适当的函数调用实现数据的读出与写入；

（5）需要考虑线程（或进程）间的同步；

（6）线程（或进程）运行结束，通过适当的系统调用结束线程（或进程）。

4 设计思路

> OS: Ubuntu 24.04.2 LTS x86_64.
>
> Language: C++.

选择“共享内存”机制实现，基于 Linux 中的 POSIX 机制，建立命名的共享内存，通过一个唯一的名称标识共享内存。

设计共享数据结构体 `SharedData` ，除了包含待排序数组的首地址，还包括一些元数据：数组大小，是否已经有序，元数据的读写锁等。使用 `pthread_mutex_t` 类型作为读写锁，保证不同子进程对共享内存的读写访问是安全的，它虽然来自线程库(POSIX Threads)，但在代码中设置为进程间共享 `pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);` ，也能实现进程的互斥访问。需要排序的数组不需要设置读写锁，因为不同子进程负责不同的数组区间，进行读写的操作互不重叠。而且不同子进程虽然只负责一片区间，但是访问操作是根据首地址访问整个数组的，如果加读写锁，反而会导致同一时间只能有一个进程对数据进程排序，其他进程只能等待，无法实现进程的并发执行，失去了并行排序的意义。

```
struct SharedData
{
    // Meta data describes the information of the shared memory.
    struct MetaData
    {
        size_t data_size;           // the number of elements
        bool is_sorted;             // whether the data is sorted
        pthread_mutex_t meta_mutex; // mutex for the meta data
    } meta;

    // Variable-length array. The size is determined by the total size.
    int data[];
};
```

设计共享内存类 `SharedMemory` ，封装对共享内存的操作，包括创建、删除、获取数组大小、获取数组首地址、对元数据加锁和解锁等方法。其中，对于元数据锁的获取和对元数据的修改，只有主进程在创建共享内存时初始化、排序完成后设置为有序时才会用到。

共享内存类的核心是构造函数和析构函数。在构造函数中，共享内存需要完成以下操作：

1. 系统调用 `shm_open` ，通过共享内存名称标识，创建一片共享内存区域。如果是主进程，则创建并初始化共享内存；如果是子进程，则打开共享内存。
2. 系统调用 `ftruncate` 设置共享内存的大小，截断到已经计算好的大小 `_shm_size` 。
3. 系统调用 `mmap` ，映射共享内存到当前进程的地址空间，获得可以指向共享数据对象的 `SharedData*` 指针。
4. 如果是主进程，初始化这块共享内存的元数据，包括数组大小、数据是否有序标志、元数据读写锁等。如果是子进程，检查拿到的这块共享内存中数组大小（位于共享数据的元数据中）是否与计算好的大小一致，否则算作整个共享内存创建失败。

在析构函数中，也需要按照顺序完成对象的销毁：销毁锁变量，系统调用 `munmap` 取消映射，系统调用 `shm_unlink` 删除共享内存。

需要注意的是，不同进程持有的是同一片共享内存，也就是最初主进程创建的共享内存，但是他们持有的共享内存的对象不同。这是因为主进程和子进程的职责不完全相同，因此需要做区分。只有最初的父进程才能创建共享内存、创建地址映射，并在排序完成后取消映射并销毁共享内存。不同的进程持有不同的共享内存类 `SharedMemory` 对象，这些不同的对象通过类中成员 `SharedData *_shm_ptr` 这个指向同一片共享内存数据的指针，实现内存共享，处理同一片内存区域。

```
class SharedMemory
{
public:
    SharedMemory(const std::string &shm_name,
                 size_t data_size,
                 bool create);

    ~SharedMemory();

    // Disable the copy constructor and assignment operator.
    SharedMemory(const SharedMemory &) = delete;
    SharedMemory &operator=(const SharedMemory &) = delete;
```

```
13
14      // Only the owner can use to unlink the shared memory and destroy the mutex.
15      void remove();
16
17      bool get_is_owner() const;
18      bool get_is_valid() const;
19      size_t get_data_size() const;
20      int *get_data_ptr() const;
21      const SharedData::MetaData &get_meta() const;
22
23      void lock_meta();
24      void unlock_meta();
25
26      void set_is_sorted(bool is_sorted);
27      bool get_is_sorted() const;
28
29  private:
30      int _shm_fd;           // the file descriptor for the shared memory
31      std::string _shm_name; // the name of the shared memory
32      SharedData *_shm_ptr;  // the pointer to the shared memory
33      size_t _shm_size;      // the size of the shared memory
34      bool _is_owner;        // whether the current process is the owner of the shared
    memory
35      bool _is_valid;        // whether the process links the shared memory successfully
36
37      void init_mutex();
38      static size_t get_shm_size(size_t data_size);
39  };
```

最后是排序器类 `Sorter` ，负责子进程的划分和快速排序的具体实现，包括当个区间内排序逻辑、划分区间的确定、生出子进程、等待子进程回收、验证排序正确性。还设置成员变量，`_active_procs_num` 记录当前活跃的子进程数量，并使用 `std::atomic<int>` 类型保证多线程安全； `_max_procs_num` 设置最大允许的子进程数量；`_threshold` 设置小于该值时不再划分子进程。

构造函数中，每一次递归划分生成的子进程，接收“共享内存名称、共享内存大小、是否为创建者”三个参数，建立自己的 `SharedMemory` 对象。每次划分出的子进程分别处理左右两个子区间，各个子进程“原地”修改共享内存中某一个区间内的数据，子进程又可重复父进程的操作继续递归划分。

`std_sort` 函数，直接使用库函数 `std::sort` 排序。在划分区间小于阈值1000的时候调用此函数。

`partition` 函数，通过左右两个指针互相比对，得到划分点的下标。

`fork_and_sort` 函数，创建子进程，子进程负责排序当前区间，并等待它后来生出的子进程全部结束，父进程负责将子进程加入子进程列表中，并将排序的进程总数加1。

`quick_sort` 函数，快速排序的主函数。若当前划分区间阈值，直接使用 `std_sort()` 排序。否则，在当前活跃进程数小于20的时候，调用 `fork_and_sort()` ，子进程排序区间左半部分，父进程递归调用快速排序 `quick_sort()` 排序区间右半部分。如果当前活跃进程数已满，则不生出子进程，而是左右两个区间各自递归调用 `quick_sort()` 排序。

最后 `soer_all()` 函数，对整个数组调用 `quick_sort()` 启动整体排序，并等待各级子进程结束后，才算排序完成。

```
class Sorter
{
public:
    Sorter(const std::string &shm_name, size_t data_size, bool create,
           int max_procs_num, int threshold);

    ~Sorter();

    void sort_all();

    bool verify_sorted() const;

private:
    SharedMemory _shm;
    std::atomic<int> _active_procs_num{1}; // The number of active processes
    const int _max_procs_num;
    const int _threshold;
    std::vector<pid_t> _child_pids; // The child process IDs

    // All the range includes the left and right index: [left, right]
    // The total range is [0, data_size - 1]
    void std_sort(int left, int right);
    int partition(int left, int right);
    pid_t fork_and_sort(int left, int right);
    void quick_sort(int left, int right);
    void wait_for_children();
};
```

完整流程：

首先使用 python 脚本生成 1,000,000 个随机整数，保存为文本文件 `data/integers.txt` 。

在主文件 `main.py` 中，主进程从文本文件中读取数据，将数据加载到缓冲区中，获取其数组大小。根据共享内存名称 `/lab2_quick_sort` 和数组大小，创建共享内存 `SharedMemory` 对象（此时主进程为唯一的创建者，需要完成初始化的工作）。将数组从缓冲区加载到共享内存的数组中。再根据共享内存名称、数组大小等信息，创建排序器 `Sorter` 对象（此时已经不是共享内存创建者），直接调用 `sort_all()` 函数开始排序。排序完成后，调用 `verify_sorted()` 函数验证排序结果是否正确，最后将排序结果写入输出文件 `data/sorted_integers.txt` 中。

5 文件结构说明

```
├── build_and_run.sh            # 编译源代码、运行可执行文件的脚本文件
├── CMakeLists.txt              # CMake 构建文件，包含编译和链接的配置
├── data
|   ├── integers.txt            # 待排序数据文件
|   └── sorted_integers.txt     # 排好序的数据文件
```

```
├── include
|   ├── shared_memory.hpp
|   └── sorter.hpp
├── report
|   ├── report.md               # 实验报告
|   └── report.pdf              # 实验报告
├── run.sh                      # 运行可执行程序的脚本文件
├── scripts
|   └── generate_data.py        # 生成随机无序数据的脚本
└── src
    ├── main.cpp
    ├── shared_memory.cpp
    └── sorter.cpp
```

6 样例测试

使用 Python 文件生成随机数保存至文本文件后，程序读取文件数据并排序，最后将排序结果写入 `data/sorted_integers.txt` 文件中。可见排序算法正确执行，排序后的数据有序。

```
Reading data from: ../data/integers.txt
The number of integers: 1000000
Sorting time: 53 ms.
The data is sorted correctly.
The sorted data is written to: ../data/sorted_integers.txt
std::sort time: 89 ms.
The sorting time of the shared memory is 59.5506% of using std::sort.
```

可见，使用共享内存方法排序，性能较好，比 C++ 标准库提供的 `std::sort` 排序还要快 40% 左右。

7 思考题

（1）你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这种选择的理由。

答：我使用共享内存是因为共享内存允许所有进程直接映射同一块物理内存，无需额外拷贝，支持对内存中任意位置的随机访问。不同进程负责不同的排序区间，并且原地排序，减少了空间资源的占用。对于大规模数据的排序，共享内存机制可以指数级的（理论上，不加 CPU 资源限制的条件下）生出子进程，各个子进程并发执行排序，效率较高，天然地符合快速排序这种递归和分治的策略。而管道机制虽然能够实现简单的字节流传输，但它是单向且顺序的，在传输百万级整数时会频繁发生数据拷贝、且受限于缓冲区大小，效率较低，难以满足排序算法高速读写且管理难度较高。

（2）你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由。

答：理论上可以解决，只是效率较低、复杂度较高。

对于管道机制：每次划分区间时，为父子进程各自准备一对匿名管道，父进程通过写管道将本次划分出来的子数组发给子进程，子进程从读管道中接收后就在本地缓冲区中执行快排，排序结束后通过另一个写管道把排序结果发回父进程。每一次递归的划分操作，父进程都只需要一次写入和一次读取，只将子区那段区间拷贝送进管道、送出管道。父进程在等待子进程完成排序时，通过轮询或阻塞在管道读取端，就能接收到已经送回的有序数组区间，父进程再将它们写回原数组。

对于消息队列机制：父进程先把划分后的数组区间封装成多个消息（包括数据、左右两端索引），用一个变量标记区分左右两块区间；子进程接收消息，执行快速排序，排序完成后将结果以另一类消息发回父进程。父进程在接收端根据消息类型和索引，按顺序将各子区间写回数组。父进程等待所有子队列消息都收回，即可完成排序。

8 实验感想

首先这个问题让我思考并清晰了进程和线程的概念。在之前的银行柜员服务问题中，我们将一个顾客或一个柜员视为一个线程，但是它们只涉及消息的交换与通信，并不涉及对同一块内存区域的读写，因此使用线程和进程解决都可。而这个问题中，我们解决的是进程间的高级通信机制，不同进程有独立的地址空间，而同一个进程中的线程，天然共享同一块地址空间，不需要也无法体现 IPC 机制，因此我们不能使用 C++ 的线程库 `#include <thread>` 解决，必须传统的 System V, POSIX 等进程相关库解决。进程拥有相对隔离的内存空间，必须通过显式的内存映射机制，将不同的进程的地址空间映射到同一块物理内存，才算实现不同进程的共享内存访问。其次，本次实验再次让我体会到了底层语言的强大（虽然使用C++编写而不是C语言），简单方便的系统调用，可以直接对内存、进程进行操作，而且不像线程库一样直接调用提供好的封装类型和函数，而是需要自己手动创建共享内存、创建地址映射，结束之后需要手动回收和销毁等等。我编写的最初版代码，在第二次运行是会提示共享内存名称在系统中已经存在，我才知道程序运行过程中创建的共享内存文件描述符会一直存在，只能手动删除，因此我在共享内存类的析构函数中显式地调用删除操作，才保证共享内存随着程序运行结束而销毁。