实验 2：高级进程间通信问题

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.实验报告

0.实验环境：

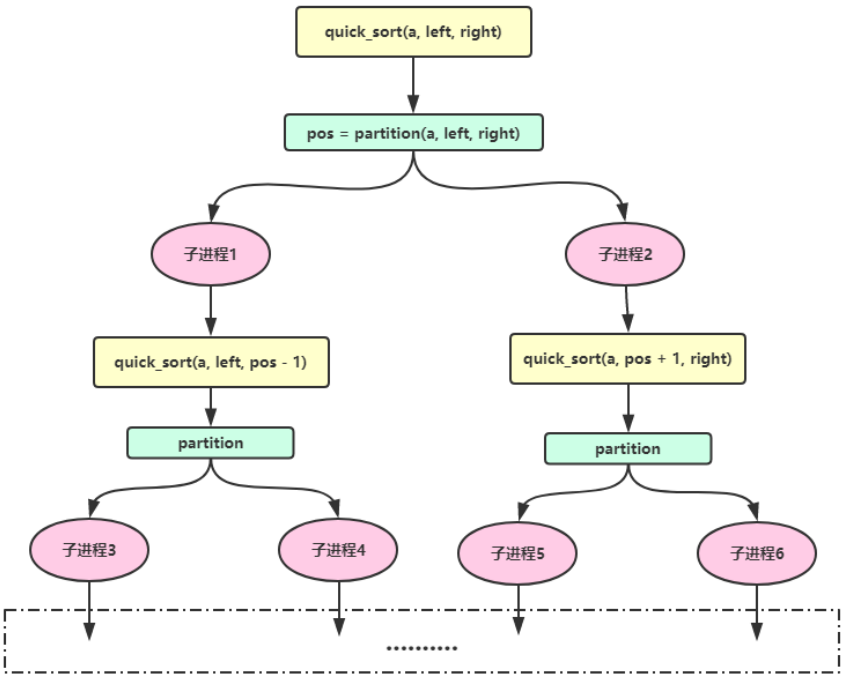
操作系统为 Windows，编程语言为 Python 与 C++。Python 用于搭建程序原型，c++程序则使用 Windows API 完成多进程排序。其中 Python 程序每次运行需要 3~4 分钟左右，请耐心等待~

a.设计思路与程序结构：

（1）Python 程序结构：

实验中，进程间通信采用 **共享内存** 机制实现。程序中通过向子进程中传递**被共享的存储随机数的数组**，来实现所谓 in-place 排序。同时由于每一次分割后产生的两个子进程不会同时读写同一数据区间，因此不需要 **互斥量** 等信号来防止同一时间多个进程修改数据。

在普通递归形式的快速排序的基础上，只需要把递归调用更改为启动新的子进程，子进程调用函数即可。具体形式如下：



为了限制子进程的数目（如果不加以限制，子进程的数量相当于递归树中节点数。即使考虑为平均二叉树，最少的子进程数为（1000000/1000）*2 - 1 = 1999 个子进程。如此多的进程，会完全耗空电脑的资源…），我设置了如下条件：1. 如果被排序数据的区间长度（即 right-left+1）大于 16000，则一次仅允许启动一个子进程。当第一个子进程完成后，才允许第二个子进程启动。2. 如果被排序数据的区间长度小于 16000，但大于 1000，则可同时启动两个子进程。3.如果被排序数据的区间长度小于 1000，则不启动新的子进程，直接排序。核心代码如下：

```
def quick_sort(a, left, right):
    #如果长度过小，直接排序，不继续分割
    if(right - left <= 1000):
        list_sort(a, left, right)
        return

    #长度小于 16000，所有进程并行操作
    elif(right - left <= 16000):
        pos = partition(a, left, right)
        #利用 共享内存 实现父进程与子进程间的通信
        cld_sort1 = mp.Process(target = quick_sort, args = (a, left, pos - 1))
        cld_sort2 = mp.Process(target = quick_sort, args = (a, pos + 1, right))
        cld_sort1.start()
        cld_sort2.start()
        cld_sort1.join()
        cld_sort2.join()
        return

    #长度过大，则只允许每个进程一次启动一个子进程，避免进程数过多
    else:
        pos = partition(a, left, right)
        #利用 共享内存 实现父进程与子进程间的通信
        cld_sort1 = mp.Process(target = quick_sort, args = (a, left, pos - 1))
        cld_sort1.start()
        cld_sort1.join()

        cld_sort2 = mp.Process(target = quick_sort, args = (a, pos + 1, right))
        cld_sort2.start()
        cld_sort2.join()
        return
```

程序中，通过调用 process.join()函数，等待子进程完成，在此之前，父进程被阻塞。**通过 join()函数来达到限制同时启动的子进程数量**。

可以看出，由于每次分叉都需要产生两个新的子进程，以上实现方式有很大的弊端：存在大量进程的开启与消亡。最少的子进程数为（1000000/1000）*2 - 1 = 1999 个子进程，

而进程的上下文切换，处理等花费大量时间。因此反而无法发挥多进程的优势，使得整个程序的运行时间达到数分钟。

因此，在 python 程序搭建的原型上，在 C++程序中使用了改进的算法，使得速度大大地提升。

（2）C++程序：

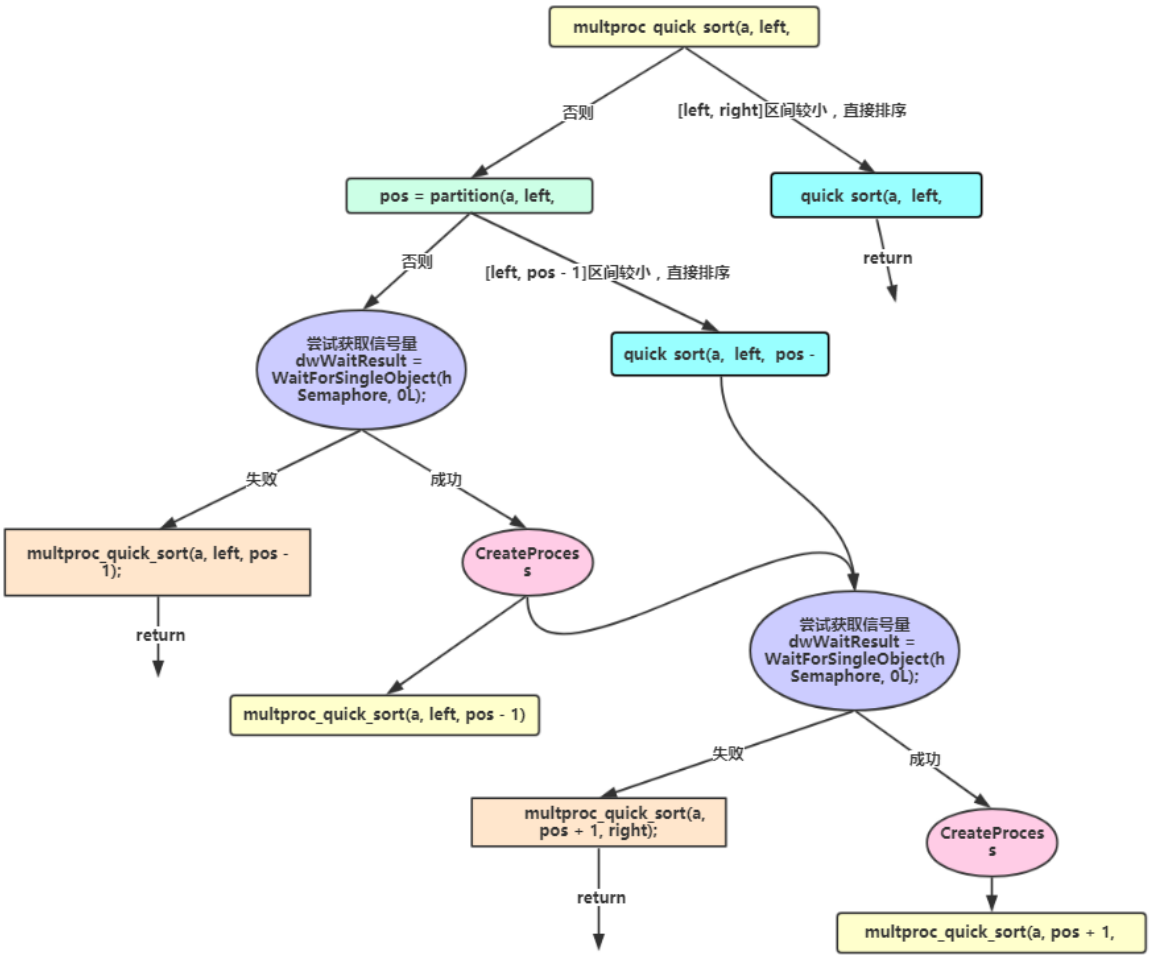
C++程序中同样使用 **共享内存** 作为进程间通信的机制，并使用 **信号量** 来控制同时开启的进程数量。多个进程间也不会同时读写同一分割区间内的数据，因此不需要 **互斥量** 等信号来防止同一时间多个进程修改数据。

在 Python 程序的基础上，我做出了如下的修改：

1. 使用 **信号量** 来控制同时开启的进程数量。每次尝试开启新进程前，父进程尝试获取信号量；如果成功获得信号量，则开启一个新的子进程用于处理分割后的子区间。这种方式可以更加灵活的控制最大进程数量。
2. 如果进程数量已达到上限，则该父进程**继续递归地往下执行**（调用相同自身函数）。在这种方式中，父进程无需等待子进程全部完成，充分利用了父进程的资源。同时也减少了需要启动的新的进程的数量。
3. 最“顶端”的进程通过类似 **轮训** 的机制判断排序是否已完成：利用 ReleaseSemaphore 函数的第三个参数，我们可以得到信号量的当前值。因此设计一循环，每隔一段时间（程序中设定为 100ms）获取并释放一次信号量，得到信号量的当前值，并判断值是否为初始设置值，即可判断排序是否已完成。主要代码如下：

```
//等待所有子进程结束
    DWORD dwWaitResult;
    LONG lpPrevCount;
    while (true) {
        dwWaitResult = WaitForSingleObject(hSemaphore, INFINITE);
        if (dwWaitResult == WAIT_OBJECT_0) {
            ReleaseSemaphore(hSemaphore, 1, &lpPrevCount);
            if (lpPrevCount == (MAX_NUM_PROC - 1)) break;
        }
        Sleep(100);
    }
```

排序程序大致流程图如下：



具体排序代码如下：

```
//多进程快速排序
void multproc_quick_sort(double *a, int left, int right, int min_part_num = 1000) {
    if (right - left <= min_part_num) { quick_sort(a, right, left); return; }
    else {
        DWORD dwWaitResult;
        wchar_t lpCommandLine1[50];
        wchar_t lpCommandLine2[50];
        STARTUPINFO si1;
        STARTUPINFO si2;
        PROCESS_INFORMATION pi1;
        PROCESS_INFORMATION pi2;

        int pos = partition(a, left, right);
        //分割后数据较少，直接排序；否则产生新的进程
        if (pos - left <= min_part_num) quick_sort(a, left, pos - 1);
        else {
            //尝试获取信号量
            dwWaitResult = WaitForSingleObject(hSemaphore, 0L);
```

```
            //可以创建新的子进程
            if (dwWaitResult == WAIT_OBJECT_0) {
                wsprintf(lpCommandLine1, L"subsort.exe %s %s %d %d", SMY_Name,
SEM_Name, left, pos - 1);
                //wcout << lpCommandLine1 << endl;
                ZeroMemory(&si1, sizeof(si1));
                si1.cb = sizeof(si1);
                ZeroMemory(&pi1, sizeof(pi1));

                if (!CreateProcess(NULL, lpCommandLine1, NULL, NULL, FALSE, 0, NULL,
NULL, &si1, &pi1))
                {
                    DWORD error_code = GetLastError();
                    cout << "- 无法创建子进程，错误码: " << error_code << " -" << '\n';
                    UnmapViewOfFile(pBuf);
                    CloseHandle(hMapFile);
                    CloseHandle(hSemaphore);
                    system("pause");
                    exit(1);
                }
            }
            //无法创建
            else {
                multproc_quick_sort(a, left, pos - 1);
                multproc_quick_sort(a, pos + 1, right);
                return;
            }
        }

        //分割后数据较少，直接排序；否则产生新的进程
        if (right - pos <= min_part_num) quick_sort(a, pos + 1, right);
        else {
            //尝试获取信号量
            dwWaitResult = WaitForSingleObject(hSemaphore, 0L);

            //可以创建新的子进程
            if (dwWaitResult == WAIT_OBJECT_0) {
                wsprintf(lpCommandLine2, L"subsort.exe %s %s %d %d", SMY_Name,
SEM_Name, pos + 1, right);
                //wcout << lpCommandLine2 << endl;
                ZeroMemory(&si2, sizeof(si2));
                si2.cb = sizeof(si2);
                ZeroMemory(&pi2, sizeof(pi2));
```

```
                    if (!CreateProcess(NULL, lpCommandLine2, NULL, NULL, FALSE, 0, NULL,
NULL, &si2, &pi2))
                    {
                        DWORD error_code = GetLastError();
                        cout << "- 无法创建子进程，错误码: " << error_code << " -" << '\n';
                        UnmapViewOfFile(pBuf);
                        CloseHandle(hMapFile);
                        CloseHandle(hSemaphore);
                        system("pause");
                        exit(1);
                    }
                }
                //无法创建
                else {
                    multproc_quick_sort(a, pos + 1, right);
                    return;
                }
            }
        }
    }
```

b.程序运行情况

（1）Python 程序：

测试中，完成整个排序过程，耗时 191.34 秒。最终正确完成排序操作。

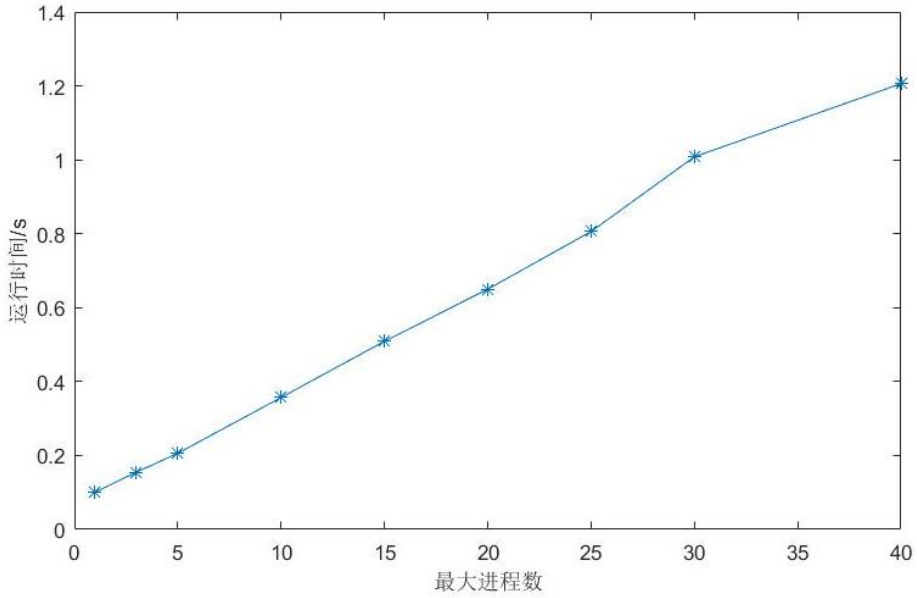
未排序的 1000000 个随机数存储在 **random_numbers.txt** 文件中；排序后的随机数存储在 **sorted_numbers_python.txt** 文件中。

(2)C++程序：

当最大进程数设置为 25 时，完成整个排序过程，耗时 0.806s。最终正确完成排序操作。

排序后的随机数存储在 **sorted_numbers_cpp.txt** 文件中。

通过修改允许的最大进程数，测量每次运行的时间（每次试验均进行四次，并将运行时间求平均），得到如下图：



可以看到：**排序的运行时间，几乎随着最大运行进程数线性增长**。具体分析见 **遇到的问题与体会。**

c.思考题解答

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：我选择了 **共享内存** 机制来完成多进程通信。因为排序过程一般是对固定的内存地址内的数据进行排序。利用共享内存，可以在不需要频繁复制数组的情况下，实现 in-place 排序，即在原内存上进行排序。

而如果使用消息队列，或者管道来实现排序，由于每个进程拥有相互独立的内存空间，通过队列或者管道传输被排序数组，数组将发生**复制**，子进程得到的数组，与父进程的数组，在内存中分布在不同位置。子进程对数组进行排序，父进程中的数组实际上未被更改。因此，如果想利用队列或者管道传递数组进行排序，需要反复拷贝数组元素的值，来实现 1.子进程与父进程数组的传递；2.同步子进程中已经被部分排序的数组和父进程中的原始数组。

综上，我选择共享内存来实现多进程快速排序。

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：另外两种机制同样可以解决问题，但是需要大量的内存复制。当父进程的某个子进

程对子序列完成排序后，父进程需要再次通过管道或者队列获得已被排序序列的拷贝，然后利用循环更新相应位置的数值即可。

d.遇到的问题与体会

1. 进程的启动，切换，结束需要消耗大量时间：

在实践中发现，Python 多进程排序速度慢于直接函数递归排序。即使对于 Python 这样运行速度很慢的动态语言，直接递归快速排序 1000000 个数据，也仅需消耗几秒钟。然而利用多进程来快速排序时，时间消耗等到 100 秒以上，增长了近两个数量级！

同样对于 C++程序，排序的**运行时间几乎随着最大运行进程数线性增长。**当最大运行进程数为 1，即不使用多进程排序数，运行时间反而最短，只需要约 0.1s。

直接递归调用函数，每次只需要进行入栈等“简单操作”；而在 Python 多进程快速排序中，假设最底层的子进程恰好对 1000 个数据直接排序，不再启动新的子进程，则在整个排序过程中，最底层的子进程（相当于二叉树的叶子节点）数量至少为 1000000/1000=1000 个，同时至少启动了 1000*2-1=1999 个进程。实际中，有的底层子进程排序的数据长度远小于 1000，这使得总的进程数量更多。

因此，**如果新的进程完成作业所需的时间**(如本实验中，排序花费的时间很少)**，与进程的启动，切换，结束消耗的时间相比不大，则使用多进程反倒可能降低程序的运行效率**。

2.在主程序中，我使用了类似 **轮训** 的方法判断排序是否完成。判断排序是否完成，同样可以使用类似 **外部中断** 的方法。利用 Windows API 中的 **事件对象**，当子进程每次释放信号量时，判断信号量的当前值是否为初始值，即所有子进程都已经结束，排序已完成。如果已完成，则利用事件对象向主进程发出消息。