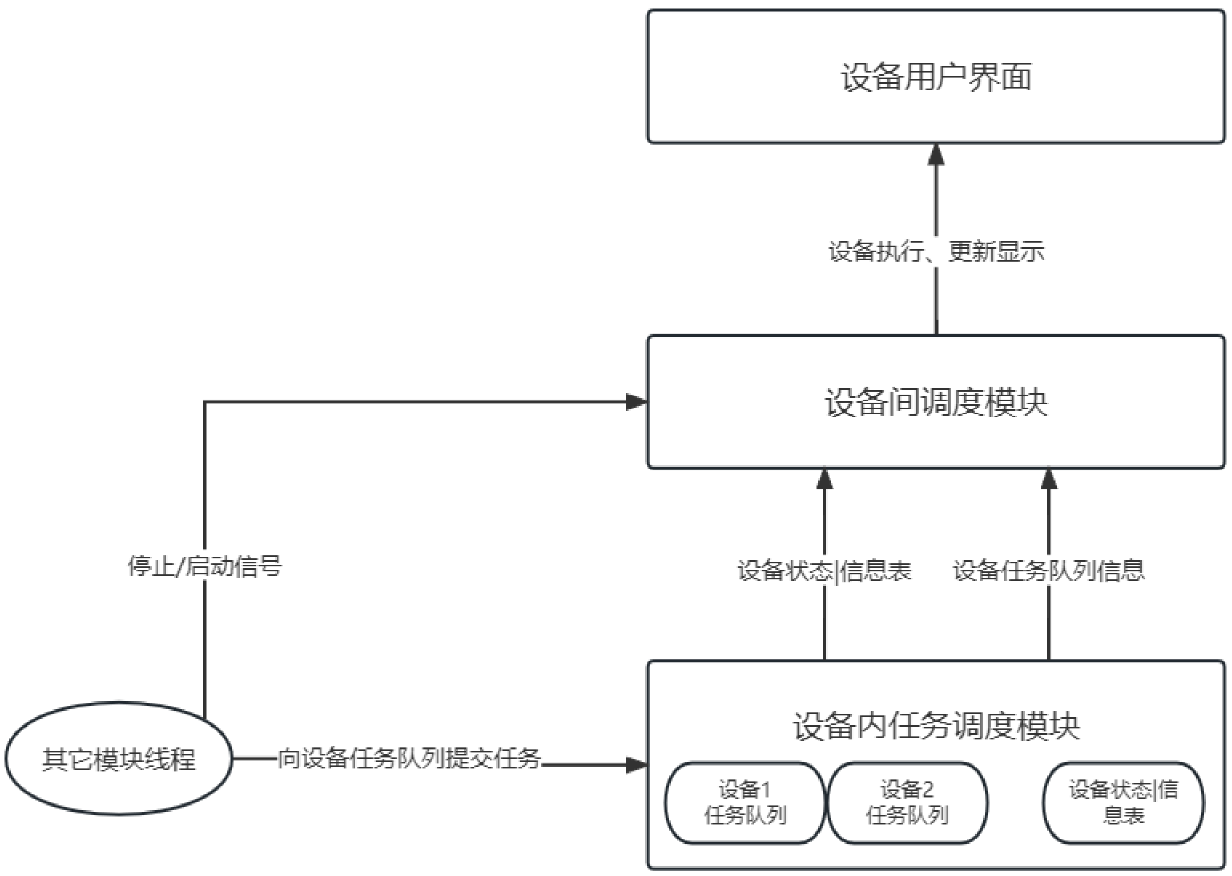
# 设备管理模块

**代码GitHub仓库地址:** xyt417/qtemu (github.com)

## *设备管理模块架构*

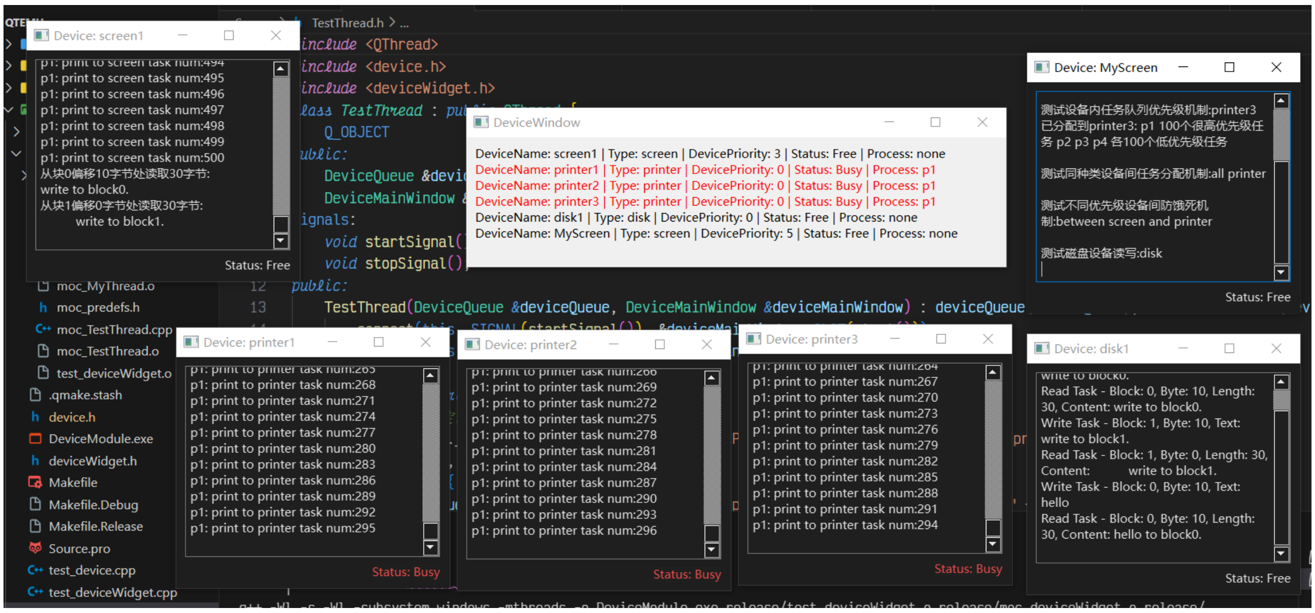


## 项目文件结构

```
文件结构：

Source文件夹：源码存放文件

Source/release: 编译输出文件

Source/release/Disk：模拟磁盘文件

Source：

- device.h: 设备信息表类 和 设备队列类 设备任务管理调度

- deviceWidget.h: 设备间调度处理模块 及 设备管理窗口和设备窗口

- MyThread.h: 测试用的线程类

- test_device: 设备信息表类 和 设备队列类 单元测试

- test_deviceWidget.cpp: 设备管理模块 模块测试

- DeviceModuleTest.exe:  设备管理模块测试 二进制可运行
```

## 设备用户界面

图为模块测试运行截图



```
Device: screen1
p1: print to screen task num:494
p1: print to screen task num:495
p1: print to screen task num:496
p1: print to screen task num:497
p1: print to screen task num:498
p1: print to screen task num:499
p1: print to screen task num:500
从块0偏移10字节处读取30字节:
write to block0.
从块1偏移0字节处读取30字节:
        write to block1.
                                Status: Free
```

```
DeviceWindow
DeviceName: screen1 | Type: screen | DevicePriority: 3 | Status: Free | Process: none
DeviceName: printer1 | Type: printer | DevicePriority: 0 | Status: Busy | Process: p1
DeviceName: printer2 | Type: printer | DevicePriority: 0 | Status: Busy | Process: p1
DeviceName: printer3 | Type: printer | DevicePriority: 0 | Status: Busy | Process: p1
DeviceName: disk1 | Type: disk | DevicePriority: 0 | Status: Free | Process: none
DeviceName: MyScreen | Type: screen | DevicePriority: 5 | Status: Free | Process: none
```

```
Device: MyScreen
测试设备内任务队列优先级机制:printer3
已分配到printer3: p1 100个很高优先级任
务 p2 p3 p4 各100个低优先级任务

测试同种类设备间任务分配机制:all printer

测试不同优先级设备间防饿死机
制:between screen and printer

测试磁盘设备读写:disk
|
                                Status: Free
```

```
Device: printer1
p1: print to printer task num:265
p1: print to printer task num:268
p1: print to printer task num:271
p1: print to printer task num:274
p1: print to printer task num:277
p1: print to printer task num:280
p1: print to printer task num:283
p1: print to printer task num:286
p1: print to printer task num:289
p1: print to printer task num:292
p1: print to printer task num:295
                                Status: Busy
```

```
Device: printer2
p1: print to printer task num:266
p1: print to printer task num:269
p1: print to printer task num:272
p1: print to printer task num:275
p1: print to printer task num:278
p1: print to printer task num:281
p1: print to printer task num:284
p1: print to printer task num:287
p1: print to printer task num:290
p1: print to printer task num:293
p1: print to printer task num:296
                                Status: Busy
```

```
Device: printer3
p1: print to printer task num:264
p1: print to printer task num:267
p1: print to printer task num:270
p1: print to printer task num:273
p1: print to printer task num:276
p1: print to printer task num:279
p1: print to printer task num:282
p1: print to printer task num:285
p1: print to printer task num:288
p1: print to printer task num:291
p1: print to printer task num:294
                                Status: Busy
```

```
Device: disk1
write to block0.
Read Task - Block: 0, Byte: 10, Length:
30, Content: write to block0.
Write Task - Block: 1, Byte: 10, Text:
write to block1.
Read Task - Block: 1, Byte: 0, Length: 30,
Content:         write to block1.
Write Task - Block: 0, Byte: 10, Text:
hello
Read Task - Block: 0, Byte: 10, Length:
30, Content: hello to block0.
                                Status: Free
```

## *接口调用说明*

接下来说明设备管理模块的接口及使用方法:

```
// 首先在主程序main中应该创建一个设备信息表并添加设备, 如:
// 设备种类有显示器screen,打印机printer,磁盘disk
// add_device(string 设备名字(随意, 认得出就行), 设备种类(以上三种之一), 设备优先级(默认为0,数字越大优先级越高))
    DeviceTable deviceTable;
    deviceTable.add_device("screen1", "screen", 1);
    deviceTable.add_device("printer1", "printer");
    deviceTable.add_device("printer2", "printer");
    deviceTable.add_device("printer3", "printer");
    deviceTable.add_device("disk1", "disk");
// 磁盘设备就创建一个就行
// 然后使用该设备信息表初始化一个设备任务队列
 DeviceQueue deviceQueue(deviceTable);
// 然后创建设备管理主窗口, 并令其显示 参数(设备信息表, 设备队列, 是否在命令行输出日志(1为是,0为否))
    DeviceMainWindow deviceMainWindow(deviceTable, deviceQueue, 1);
    deviceMainWindow.show();
```

初始化完成后就可以在其他线程中使用设备管理模块了

这里创建了一个线程，并在该线程中调用使用为例：

```
class TestThread : public QThread {
    Q_OBJECT //使用Q_OBJECT 从而能够使用Qt信号与槽通信
private:
    DeviceQueue &deviceQueue;
    DeviceMainWindow &deviceMainWindow;
signals: // 定义两个信号用于启动设备处理，和停止设备处理
    void startSignal();
    void stopSignal();
public:
     MyThread(DeviceQueue &deviceQueue, DeviceMainWindow
&deviceMainWindow) : deviceQueue(deviceQueue),
deviceMainWindow(deviceMainWindow) {
         // 将该信号连接到deviceMainWindow的start()和stop函数
        connect(this, SIGNAL(startSignal()),
&deviceMainWindow, SLOT(start()));
        connect(this, SIGNAL(stopSignal()),
&deviceMainWindow, SLOT(stop()));
    }
    void run() override {
        emit stopSignal(); // emit该信号即可时设备处理程序暂停
        emit startSignal();  // 或启动
        // 不需要可以不用，设备处理程序默认启动

        // 接口:
        // deviceQueue.allocate_device(string设备种类，string
进程名称，string任务信息，int任务在该设备队列中的优先级【且数字越
大优先级越高】)
        // 目前支持的任务种类: print任务(只支持 显示器screen，和
打印机printer)、read/write任务(只支持 磁盘disk)
        // 任务信息中参数以逗号分隔，磁盘块号范围为0~99，共100个
磁盘块
        // print任务:"print,text(要打印的字符串)"
        // write任务:"write,磁盘快号,距磁盘块首部偏移字节
数,text(写入的字符串)"
```

```
        // read任务:"read,磁盘快号,距磁盘块首部偏移字节数,读取长
度,读取到目标buffer号(1~999)"
        // 示例:
            // 打印到打印机
            deviceQueue.allocate_device("printer", "p1",
"print,p1: hello printer i'm p1");
            deviceQueue.allocate_device("printer", "p2",
"print,p2: hello printer i'm p2");
            // 打印到显示器
            deviceQueue.allocate_device("screen", "p3",
"print,p3: hello printer i'm p3", 2);
            // 读取和写入
            deviceQueue.allocate_device("disk", "p0",
"write,0,10,hello disk0.");
            deviceQueue.allocate_device("disk", "p1",
"read,0,10,12,0");
            // 查看读取到的数据
            if(!deviceQueue.readInBuffer[0].empty())
                cout << deviceQueue.readInBuffer[0]; // index
为 0~999
       // 如果确定要分配的设备名字，要将任务分配给该指定设备:
       // 使用：deviceQueue._allocate(string设备名, string进程
名称, string任务信息, int任务在该设备队列中的优先级【且数字越大优
先级越高】)
        // 示例:
            deviceQueue._allocate_device("HuaWeiPrinter",
"p1", "print,p1: hello HuaWeiPrinter");
    }
}
```

## *主要数据结构概览*

## 设备任务队列

```cpp
class DeviceQueue {
private:
    DeviceTable &deviceTable;                                // 设备信息表

    vector<string> devices;                                  // 设备名列表
    vector<string> types;                                    // 设备类型列表
    vector<string> available_devices;                        // 空闲设备列表
    map<string, vector<DevRequest>> occupied_devices;   // 正在使用的设备字典，键为设备名，值为使用该设备的进程列表

    // 主要方法：
    // 分配给具体设备任务
    bool _allocate_device(string device_name, string process_name, string request, int priority = 0);
    // 分配给某一特定类型设备任务
    bool allocate_device(string device_type, string process_name, string request = "", int priority = 0);
    // 确认执行完某一任务，释放设备并将其移出任务队列
    bool release_device(string device_name, string &process_name);

}
```

## 设备信息表

```
class DeviceTable {
private:
    int devNum;                    // 设备数量
    vector<Device> deviceList;   // 设备列表

    // 主要方法:
    // 添加设备
    bool add_device(string name, string type, int priority = 
0);
    // 删除设备
    bool remove_device(string name);
    // 更新设备状态
    bool change_device_status(string name, int status, string 
pname);
}
```

## 任务请求

```
struct DevRequest {
    string pname;         // 进程名
    string requestStr;   // 任务请求信息
    int priority;         // 任务优先级
};
```

## 设备信息块

```
struct Device {
    string name;     // 设备名
    string type;     // 设备类型
    int status;      // 设备状态
    string pname;    // 占用或将占用的进程
    int priority;    // 设备优先级
};
```

# *设备调度算法说明*

## 设备间调度

设备处理模块每一个处理周期处理一个设备任务队列中的一个任务。

#### 相同优先级设备

以相同优先级的设备视角来看，设备处理模块在属于该优先级设备的处理周期依次遍历每一个同优先级的设备，实现同优先级设备分配相同的处理资源

#### 不同优先级设备

每一个设备拥有一个整形变量表示优先级，数字最小为0(默认值)，数字越大优先级越高。对于设备优先级为x的设备集合，该集合所有设备在所有更低优先级设备处理完任务或用尽它们的周期之前，获得的处理周期数不超过2^x个周期。当该集合设备任务为空或连续获得的处理周期数达到2^x个周期时，模块将优先处理还未使用完分配周期数的更低优先级的设备集合。在高优先级设备获得的处理周期数用完之前，接到任务时还可以抢占接下来的处理周期，否则不能抢占。当所有优先级设备集合都用尽分配周期或任务为空时将重新分配处理周期。

通过该调度方法，可以实现相同优先级设备获得相同处理资源，优先级更高设备获得更多处理资源，同时低优先级设备不会因得不到处理资源而饿死。

## 设备内任务调度

设备内任务队列在该设备获得一个处理周期时，提供并释放一个任务。

### 相同优先级任务

对于设备收到的相同优先级任务，采用FCFS方案，实现相同优先级设备资源公平分配。

### 不同优先级任务

每一个任务拥有一个整形变量表示优先级，数字最小为0(默认值)，数字越大优先级越高。对于设备收到的不同优先级任务，将优先执行优先级最高的任务，当设备执行了一定数量的任务时，若设备任务队列内还有任务没有执行，则将设备任务队列内还未执行的所有设备优先级+1。

通过该调度方法，可以实现相同优先级任务获得公平的处理资源分配，足够高优先级的任务可以优先执行，同时在不断有高优先级任务申请使用设备时，低优先级的任务也不会因无法得到执行而饿死。

## 任务分配

### 指定设备分配

进程可以通过指定设备名称将任务分配给指定设备

### 同种设备分配

进程可以指定设备种类，将任务分配给同一类设备。如同时有多个同种设备时，系统会将任务分配给该类设备集合中当前任务队列最短的设备。实现任务的均匀分配，从而充分利用设备资源。

# 模块测试代码

```
#include <QThread>
#include <device.h>
#include <deviceWidget.h>
class TestThread : public QThread {
    Q_OBJECT
public:
    DeviceQueue &deviceQueue;
    DeviceMainWindow &deviceMainWindow;
signals:
    void startSignal();
    void stopSignal();
public:
    TestThread(DeviceQueue &deviceQueue, DeviceMainWindow
&deviceMainWindow) : deviceQueue(deviceQueue),
deviceMainWindow(deviceMainWindow) {
        connect(this, SIGNAL(startSignal()),
&deviceMainWindow, SLOT(start()));
        connect(this, SIGNAL(stopSignal()),
&deviceMainWindow, SLOT(stop()));
    }
    void run() override {
    // 测试分配给指定设备
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,测试分配给指定设备:printer1");
        int n = 100, m = 100;
        while(n --){
            deviceQueue._allocate_device("printer1", "p1",
"print,p1: hello printer num:" + to_string(m - n));
            deviceQueue._allocate_device("printer1", "p2",
"print,p2: hello printer num:" + to_string(m - n));
            deviceQueue._allocate_device("printer1", "p3",
"print,p3: hello printer num:" + to_string(m - n));
        }
        QThread::msleep(5000);
    // 测试设备内任务队列防饿死机制
```

```
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,\n测试设备内任务队列防饿死机
制:printer2");
        emit stopSignal(); // 暂停设备处理
        // 初始分配 100 个高优先级任务 和 100 个低优先级任务
        n = 100, m = 100;
        while(n --){
            deviceQueue._allocate_device("printer2", "p1",
"print,p1: higher priority num:" + to_string(m - n), 1); //
较高优先级任务
            deviceQueue._allocate_device("printer2", "p2",
"print,p2: lower priority num:" + to_string(m - n), 0); // 较
低优先级任务
        }
        emit startSignal(); // 启动设备处理
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,已分配到printer2: 100个较高优先级任务 100
个较低优先级任务");
        QThread::msleep(5000);
    // 测试任务优先机制
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,\n测试设备内任务队列优先级机
制:printer3");
        n = 100, m = 100;
        while(n --){
            deviceQueue._allocate_device("printer3", "p1",
"print,p1: p1(high) task num:" + to_string(m - n), 10); // 高
优先级任务
            deviceQueue._allocate_device("printer3", "p2",
"print,p2: p2(low) task num:" + to_string(m - n)); // 低优先级
任务
            deviceQueue._allocate_device("printer3", "p3",
"print,p3: p3(low) task num:" + to_string(m - n)); // 低优先级
任务
            deviceQueue._allocate_device("printer3", "p4",
"print,p4: p4(low) task num:" + to_string(m - n)); // 低优先级
任务
        }
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,已分配到printer3: p1 100个很高优先级任务
p2 p3 p4 各100个低优先级任务");
        QThread::msleep(10000);
    // 测试同种类设备间任务分配
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,\n测试同种类设备间任务分配机制:all
printer");
        n = 500, m = 500;
        while(n --){
            deviceQueue.allocate_device("printer", "p1",
"print,p1: printer task num:" + to_string(m - n));
        }
        QThread::msleep(10000);
    // 测试不同优先级设备间防饿死机制
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,\n测试不同优先级设备间防饿死机制:between
screen and printer");
        n = 500, m = 500;
        while(n --){
            deviceQueue.allocate_device("printer", "p1",
"print,p1: print to printer task num:" + to_string(m - n));
            deviceQueue._allocate_device("screen1", "p1",
"print,p1: print to screen task num:" + to_string(m - n));
        }
        QThread::msleep(10000);
    // 磁盘设备读写测试
        deviceQueue._allocate_device("MyScreen",
"TestProcess", "print,\n测试磁盘设备读写:disk");

        deviceQueue.allocate_device("disk", "p0",
"write,0,10,write to block0.", 3);
        deviceQueue.allocate_device("disk", "p1",
"read,0,10,30,0", 3);
        QThread::msleep(1000);
        deviceQueue._allocate_device("screen1", "p1", "print,
从块0偏移10字节处读取30字节:");
        string readInStr = deviceQueue.readInBuffer[0];
        deviceQueue._allocate_device("screen1", "p1",
"print," + readInStr);

        deviceQueue.allocate_device("disk", "p0",
"write,1,10,write to block1.", 3);
        deviceQueue.allocate_device("disk", "p1",
"read,1,0,30,1", 3);
        QThread::msleep(1000);
        deviceQueue._allocate_device("screen1", "p1", "print,
从块1偏移0字节处读取30字节:");
        readInStr = deviceQueue.readInBuffer[1];
        deviceQueue._allocate_device("screen1", "p1",
"print," + readInStr);

        deviceQueue.allocate_device("disk", "p0",
"write,0,10,hello", 3);
        deviceQueue.allocate_device("disk", "p1",
"read,0,10,30,2", 3);
        QThread::msleep(1000);
        deviceQueue._allocate_device("screen1", "p1", "print,
从块0偏移10字节处读取30字节:");
        readInStr = deviceQueue.readInBuffer[2];
        deviceQueue._allocate_device("screen1", "p1",
"print," + readInStr);

    }
};
```

**模块代码:**

# 设备内任务调度模块

```
// device.h:
#ifndef DEVICE
#define DEVICE

#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <algorithm>
#include <QMutex>


using namespace std;

#define FREE 0
#define BUSY 1

#define NOEXIST "noexist"
#define EMPTY "empty"


struct DevRequest {
    string pname;       // 进程名
    string requestStr;  // 任务信息
    int priority;       // 任务优先级
};

struct Device {
    string name;
    string type;
    int status;
    string pname;   // 占用或将占用的进程
    int priority;   // 设备优先级
};
```

```
// 设备信息表
class DeviceTable {
public:
    QMutex mutex2; // 加个锁不知道会不会出现bug
public:
    int devNum;
    vector<Device> deviceList;

public:
    DeviceTable() : devNum(0) {}

    Device operator[](string name){
        // 根据设备名获取设备信息
        for (vector<Device>::iterator it =
deviceList.begin(); it != deviceList.end(); ++ it) {
            if (it->name == name) {
                return *it;
            }
        }
        Device temp {"none", "none", 0, "none"};
        return temp;
    }

    bool add_device(string name, string type, int priority =
0) {
        // 不允许重名设备
        for (vector<Device>::iterator it =
deviceList.begin(); it != deviceList.end(); ++ it) {
            if (it->name == name) {
                return false;
            }
        }
        Device new_device {name, type, FREE, "none",
priority};
        deviceList.push_back(new_device);
        devNum ++;
        return true;
    }
    bool remove_device(string name) {
        for (vector<Device>::iterator it =
deviceList.begin(); it != deviceList.end(); ++ it) {
            if (it->name == name) {
                deviceList.erase(it);
                devNum --;
                return true;
            }
        }
        return false;
    }

    bool change_device_status(string name, int status, string
pname) {
        for (vector<Device>::iterator it =
deviceList.begin(); it != deviceList.end(); ++ it) {
            if (it->name == name) {
                it->status = status;
                if(status == FREE)
                    it->pname = "none";
                else
                    it->pname = pname;
                return true;
            }
        }
        return false;
    }

    vector<Device> get_device_list() {
        mutex2.lock();
        vector<Device> deviceListCopy = deviceList;
        mutex2.unlock();
        return deviceListCopy;
    }

    void get_names(vector<string> &names) {
        for (vector<Device>::iterator it =
deviceList.begin(); it != deviceList.end(); ++ it) {
            names.push_back(it->name);
        }
    }

    void get_types(vector<string> &types) {
        for (vector<Device>::iterator it =
deviceList.begin(); it != deviceList.end(); ++ it) {
            types.push_back((*it).type);
        }
        sort(types.begin(), types.end());
        vector<string>::iterator new_end =
unique(types.begin(), types.end());
        types.erase(new_end, types.end());
    }

    int dev_num() {
        return devNum;
    }

    void printInfo() {
        cout << "|DeviceName" << "|Type" << "|priority" <<
"|Status" << "|process|\n";
        for (auto it : deviceList) {
            cout << "  " << it.name << "  " << it.type << "
" << it.priority << "  " << (it.status ? "busy" : "free") <<
"  " << it.pname << "  \n";
        }
    }
};


// 设备队列类
class DeviceQueue {
public:
    DeviceTable &deviceTable;                           // 设
备信息表
```

```
    vector<string> devices;                               // 设
备名列表
    vector<string> types;                                 // 设
备类型列表
    vector<string> available_devices;                     // 空
闲设备列表
    map<string, vector<DevRequest>> occupied_devices;   // 正
在使用的设备字典，键为设备名，值为使用该设备的进程列表

    map<string, int> countRecord;                           //
防饿死机制

    QMutex mutex1; // 加个锁不知道会不会出现bug

public:
    DeviceQueue(DeviceTable &_deviceTable) :
deviceTable(_deviceTable) {
        //初始化空闲设备列表
        deviceTable.get_names(devices);
        for(string device : devices){
            available_devices.push_back(device);
        }
        deviceTable.get_types(types);
        // for (auto it : types) cout << it << " ";

        // 初始化 buffer
        readInBuffer = vector<string>(1000, "");
    }

    // 分配设备给进程 (设备类型，进程名称，任务信息，任务优先级)
    bool _allocate_device(string device_name, string
process_name, string request, int priority = 0) {
        // 如果设备不存在，则返回 false
        vector<string>::iterator it = find(devices.begin(),
devices.end(), device_name);
        if (it == devices.end()) {
            return false;
        }

        mutex1.lock();

        // 如果设备已经被使用，则将进程添加到设备的使用列表中
        if (occupied_devices.find(device_name) !=
occupied_devices.end()) {
            // 防饿死机制
            if(countRecord[device_name] % 100 == 0){
                countRecord[device_name] = 0;
                for(auto it =
occupied_devices[device_name].begin(); it !=
occupied_devices[device_name].end(); ++ it){
                    ++ (it->priority);
                }
            }
            countRecord[device_name] ++;
            if(priority == 0 ||
occupied_devices[device_name].size() == 1) // 默认优先级或只有
一个处理中的任务，直接添加到队列尾部

 occupied_devices[device_name].push_back(DevRequest{process_n
ame, request, priority});
            else{
                // 非默认优先级，添加到队列中合适的位置
                vector<DevRequest>& processes =
occupied_devices[device_name];
                auto it = processes.begin() + 1;
                for(; it != processes.end(); ++ it){
                    if(it->priority < priority){
                        processes.insert(it,
DevRequest{process_name, request, priority});
                        break;
                    }
                }
                if(it == processes.end()) // 全为同一种高优先
级，it到达队尾
```

```
 processes.push_back(DevRequest{process_name, request,
priority});
            }
        }
        // 如果设备未被使用，则创建一个新的使用列表并添加进程，并
将该设备移出空闲列表
        else {
            it = find(available_devices.begin(),
available_devices.end(), device_name);
            available_devices.erase(it);
            occupied_devices[device_name] =
vector<DevRequest>{};

 occupied_devices[device_name].push_back(DevRequest{process_n
ame, request, priority});
            countRecord[device_name] = 1; // 防饿死机制 初始化
            deviceTable.change_device_status(device_name,
BUSY, process_name);
        }
        mutex1.unlock();
        return true;
    }

    // 分配设备给进程 (设备类型，进程名称，任务信息，任务优先级)
    bool allocate_device(string device_type, string
process_name, string request = "", int priority = 0) {
        // 不存在该类型设备
        vector<string>::iterator itt = find(types.begin(),
types.end(), device_type);
        if (itt == types.end()) {
            return false;
        }
        // 存在空闲设备
        for(auto it = available_devices.begin(); it !=
available_devices.end(); ++ it) {
            if(deviceTable[*it].type == device_type) {
                return _allocate_device(*it, process_name,
request, priority);
            }
        }

        // 不存在空闲设备，找到任务队列最短设备
        long long unsigned int len = 99999;
        string device_name;
        auto it = occupied_devices.begin();
        for(; it != occupied_devices.end(); ++ it) {
            if(deviceTable[(*it).first].type == device_type
&& (*it).second.size() < len) {
                len = (*it).second.size();
                device_name = (*it).first;
            }
        }

        return _allocate_device(device_name, process_name,
request, priority);
    }

    // 释放设备 (设备名称，确认结束任务对应进程名，备用)
    bool release_device(string device_name, string
&process_name) {
        // 如果设备不存在，返回 false
        vector<string>::iterator it = find(devices.begin(),
devices.end(), device_name);
        if (it == devices.end()) {
            process_name = NOEXIST;
            return false;
        }

        // 如果该设备未被占用，返回 flase
        if (occupied_devices.find(device_name) ==
occupied_devices.end()){
            process_name = EMPTY;
            return false;
        }
        mutex1.lock();

        // 将进程从设备的使用列表中删除
        vector<DevRequest>& processes =
occupied_devices[device_name];
        process_name = (*processes.begin()).pname;
        processes.erase(processes.begin());

        // 如果设备的使用列表为空，则将设备添加回空闲设备列表中
        if (processes.empty()) {
            available_devices.push_back(device_name);
            occupied_devices.erase(device_name);
            deviceTable.change_device_status(device_name,
FREE, "");
        } else {
            deviceTable.change_device_status(device_name,
BUSY, (*processes.begin()).pname);
        }
        mutex1.unlock();
        return true;
    }

    // 获取可用设备列表
    vector<string> get_available_devices() {
        mutex1.lock();
        vector<string> available_devices_copy =
available_devices;
        mutex1.unlock();
        return available_devices_copy;
    }

    // 获取正在使用设备的字典
    map<string, vector<DevRequest>> get_occupied_devices() {
        mutex1.lock();
        map<string, vector<DevRequest>> occupied_devices_copy
= occupied_devices;
        mutex1.unlock();
        return occupied_devices_copy;
    }

    // 打印可用设备列表
    void print_avaliable_devices() {
        vector<string> available_devices =
get_available_devices();
        cout << "Available Devices: ";
        for (string device_name : available_devices) {
            cout << device_name << " ";
        }
        cout << '\n';
    }

    // 打印正在使用设备的字典
    void print_occupied_devices() {
        map<string, vector<DevRequest>> occupied_devices =
get_occupied_devices();
        cout << "Occupied Devices: " << '\n';
        for (auto& pair : occupied_devices) {
            string device_name = pair.first;
            vector<DevRequest> processes = pair.second;
            cout << device_name << ": ";
            for (DevRequest process : processes) {
                cout << process.pname << "[\"" <<
process.requestStr << "\"" << ":" << process.priority << "]"
<< " ";
            }
            cout << '\n';
        }
    }

public:
    // 设置1000个buffer
    vector<string> readInBuffer;
};

#endif
```

# 设备间任务调度模块 及 设备用户界面

```
// deviceWidget.h:
#ifndef DEVICE_WIDGET
#define DEVICE_WIDGET
#include <QApplication>
#include <QMainWindow>
#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>
#include <QString>
#include <QTimer>
#include <QMap>
#include <QThread>
#include <QTextEdit>
#include <QPlainTextEdit>
#include <QScrollBar>
#include <QStyle>

#include <fstream>

#include <device.h>

const int DISK_BLOCK_SIZE = 4096;  //定义磁盘块大小 (4096
Byte)
const int DISK_BLOCK_NUM = 100;     //定义磁盘块数量 (100
Blocks)

const int MAX_DEVICE_PRIORITY = 10; // 最大优先级 running_v2

#define SUCCESS 0
#define FILE_ERR 1
#define OVERSTEP 2

// 设备类
class DeviceWindow : public QMainWindow {
    Q_OBJECT
```

```
public:
    explicit DeviceWindow(QString deviceType = "none",
QString deviceName = "none", QWidget *parent = nullptr) :
QMainWindow(parent) {
        setWindowTitle("Device: " + deviceName);
        centralWidget = new QWidget(this);
        setCentralWidget(centralWidget);
        layout = new QVBoxLayout(centralWidget);

        // 显示部件
        printerWidget = new QTextEdit(centralWidget);
        layout->addWidget(printerWidget);
        // 状态标签
        statusLabel = new QLabel("Status: Free", this);
        statusLabel->setAlignment(Qt::AlignBottom |
Qt::AlignRight);
        layout->addWidget(statusLabel);

        layout->setAlignment(statusLabel, Qt::AlignBottom |
Qt::AlignRight);
    }

    void setStatus(bool isBusy) {
        statusLabel->setText("Status: " + QString(isBusy ?
"Busy" : "Free"));
        if(isBusy) statusLabel->setStyleSheet("background-
color: rgba(31, 31, 31, 180); color: rgb(241, 76, 76);");
        else statusLabel->setStyleSheet("background-color:
rgba(31, 31, 31, 180); color: rgb(230, 230, 230);");
    }

    void print(const QString& text) {
        printerWidget->insertPlainText(text + '\n');
        printerWidget->moveCursor(QTextCursor::End);
        printerWidget->verticalScrollBar()-
>setValue(printerWidget->verticalScrollBar()->maximum());
    }
    int writeToFile(int blockIndex, int byteIndex, string
text) {
        if(blockIndex < 0 || blockIndex >= DISK_BLOCK_NUM)
return OVERSTEP; // 磁盘块索引越界
        if(byteIndex < 0 || byteIndex + text.size() >=
DISK_BLOCK_SIZE) return OVERSTEP; // 字节索引越界
        ofstream outfile("release/Disk/block" +
to_string(blockIndex) + ".txt", std::ios::binary |
std::ios::in | std::ios::out);
        if(!outfile.is_open()) return FILE_ERR; // 文件打开失
败
        outfile.seekp(byteIndex, ios::beg); // 定位到第
byteIndex个字节  ios::beg: 文件开头
        outfile << text; // 写入text
        outfile.close();
        return SUCCESS;
    }

    int readFromFile(int blockIndex, int byteIndex, int
length, string &text) {
        if(blockIndex < 0 || blockIndex >= DISK_BLOCK_NUM)
return OVERSTEP; // 磁盘块索引越界
        if(byteIndex < 0 || byteIndex + length >=
DISK_BLOCK_SIZE) return OVERSTEP; // 字节索引越界
        ifstream infile("release/Disk/block" +
to_string(blockIndex) + ".txt", std::ios::binary);
        if(!infile.is_open()) return FILE_ERR; // 文件打开失败
        infile.seekg(byteIndex, ios::beg); // 定位到第
byteIndex个字节  ios::beg: 文件开头
        text.clear(); // 清空text
        text.resize(length); // 重置text大小
        infile.read(&text[0], length); // 读取length个字节
        infile.close();
        return SUCCESS;
    }

public:
    QWidget *centralWidget; // 中心窗口
    QVBoxLayout *layout; // 布局管理器
    QTextEdit *printerWidget; // 打印机组件
    QLabel *statusLabel; // 状态标签

};


// 主窗口类
class DeviceMainWindow : public QMainWindow {
    Q_OBJECT

public:
    explicit DeviceMainWindow(DeviceTable &deviceTable,
DeviceQueue &deviceQueue, int logger = 0, QWidget *parent =
nullptr)
      : QMainWindow(parent), deviceTable(deviceTable),
deviceQueue(deviceQueue), logger(logger) {
        setWindowTitle("DeviceWindow");
        centralWidget = new QWidget(this);
        setCentralWidget(centralWidget);
        layout = new QVBoxLayout(centralWidget);
        statusLabel = new QLabel(centralWidget);
        layout->addWidget(statusLabel);

        // 创建设备窗口
        createDeviceWindows();
        updateDeviceStatus();

        // 初始化设备指针
        occupied_devices =
deviceQueue.get_occupied_devices();
        devicePointer = occupied_devices.begin();

        // 初始化运行记录 running_v2
        resetRunningRecord();

        // 设置定时器
        timer = new QTimer(this);
        connect(timer, SIGNAL(timeout()), this,
SLOT(running()));
        timer->start(10); // 处理频率 10ms

    }

    void enableLogger() {logger = 1;}
    void disableLogger() {logger = 0;}

    void resetRunningRecord() {
        for(int i = 0; i <= MAX_DEVICE_PRIORITY; ++ i) {
            running_record[i] = pow(2, i);
        }
    }

    string argi(string request, int index){ // 请求的第 i 个参
数
        int i = 1;
        int start = 0;
        int end = request.find(",");
        while(i < index){
            start = request.find(",", start) + 1;
            end = request.find(",", start);
            ++ i;
        }
        return request.substr(start, end - start);
    }

public slots:

void updateDeviceStatus() {
    // 更新设备状态显示
    QString statusText;
    vector<Device> deviceList =
deviceTable.get_device_list();
    for (const auto &device : deviceList) {
        QString deviceStatus = device.status ? "Busy" :
"Free";
        QString statusLine = QString("DeviceName: %1  |
Type: %2  |  DevicePriority: %3  |  Status: %4  |  Process:
%5")

.arg(QString::fromStdString(device.name))

.arg(QString::fromStdString(device.type))
                                .arg(device.priority)
                                .arg(deviceStatus)

.arg(QString::fromStdString(device.pname));

        // 创建带样式的文本
        QString styledLine = device.status ? ("<font
color='red'>" + statusLine + "</font>") : statusLine;

        statusText += styledLine + "<br>";  // 添加换行符

        if (device.type == "screen") {
            // 更新屏幕设备窗口的状态
            screenWindows[device.name]-
>setStatus(device.status);
        } else if (device.type == "printer") {
            // 更新打印机设备窗口的状态
            printerWindows[device.name]-
>setStatus(device.status);
        } else if (device.type == "disk") {
            // 更新磁盘设备窗口的状态
            diskWindows[device.name]-
>setStatus(device.status);
        }
    }
    statusLabel->setText(statusText);
}

    void running(){ // running_v2
```

```
        // 更新occupied_devices
        if(devicePointer == occupied_devices.end()){
            occupied_devices = 
deviceQueue.get_occupied_devices();
            devicePointer = occupied_devices.begin();
            // 更新max_priority
            int _max_priority = 0;
            for(auto device : occupied_devices){
                int priority = 
deviceTable[device.first].priority;
                if(priority > _max_priority) _max_priority = 
priority;
            }
            max_priority = _max_priority;
            // 防饿死机制 running_v2
            while(running_record[max_priority] == 0){
                -- max_priority;
                if(max_priority == -1) {
                    resetRunningRecord();
                    max_priority = _max_priority;
                }
                // else if(logger) cout << "防饿死机制: 优先级
" << max_priority + 1 << " 的设备不执行\n";
            }
            -- running_record[max_priority];
        }
        // 遍历occupied_devices
        while(devicePointer != occupied_devices.end() && 
deviceTable[devicePointer->first].priority != max_priority)
            ++ devicePointer; // 优先级不是最高的设备不执行
        if(devicePointer == occupied_devices.end()) return;
        string device_name = devicePointer->first;
        vector<DevRequest> requests = devicePointer->second;
        string process_name = requests[0].pname;
        string request = requests[0].requestStr;
        // screen
        if(deviceTable[device_name].type == "screen"){
            // request = "print,text"
            // 屏幕打印
            if(request.find("print") != string::npos){
                if(logger) cout << "设备 " << device_name <<
" 执行进程 " << process_name << " 的任务:[" << request <<
"]\n";
                screenWindows[device_name]-
>print(QString::fromStdString(argi(request, 2)));
            }
        // printer
        }else if(deviceTable[device_name].type == "printer"){
            // request = "print,text"
            // 打印机打印
            if(request.find("print") != string::npos){
                if(logger) cout << "设备 " << device_name <<
" 执行进程 " << process_name << " 的任务:[" << request <<
"]\n";
                printerWindows[device_name]-
>print(QString::fromStdString(argi(request, 2)));
            }
        // disk
        }else if(deviceTable[device_name].type == "disk"){
            if (request.find("write") != std::string::npos) {
                // 写入磁盘
                // request =
"write,blockIndex,byteIndex,text"
                int blockIndex = std::stoi(argi(request, 2));
                int byteIndex = std::stoi(argi(request, 3));
                string text = argi(request, 4);

                int flag;
                if ((flag = diskWindows[device_name]-
>writeToFile(blockIndex, byteIndex, text)) == SUCCESS) {
                    if (logger)
                        std::cout << "设备 " << device_name
<< " 执行进程 " << process_name << " 的任务:[" << request <<
"]\n";

                    // 在窗口中显示任务信息
                    QString taskInfo = "Write Task - Block: "
+ QString::number(blockIndex) +
                                    ", Byte: " +
QString::number(byteIndex) +
                                    ", Text: " +
QString::fromStdString(text);
                    diskWindows[device_name]-
>print(taskInfo);
                } else if (flag == OVERSTEP) {
                    // 处理越界情况
                    if (logger)
                        std::cout << "设备 " << device_name
<< " 执行进程 " << process_name << " 的任务越界:[" << request
<< "]\n";

                    // 在窗口中显示越界信息
                    QString errorInfo = "Error: OverStep -
Block: " + QString::number(blockIndex) +
                                        ", Byte: " +
QString::number(byteIndex) +
                                        ", Text: " +
QString::fromStdString(text);
                    diskWindows[device_name]-
>print(errorInfo);
                } else if (flag == FILE_ERR) {
                    // 处理文件打开失败情况
                    if (logger)
                        std::cout << "设备 " << device_name
<< " 执行进程 " << process_name << " 的任务文件打开失败:[" <<
request << "]\n";

                    // 在窗口中显示文件打开失败信息
                    QString errorInfo = "Error: File Open
Failed - Block: " + QString::number(blockIndex) +
                                        ", Byte: " +
QString::number(byteIndex) +
                                        ", Text: " +
QString::fromStdString(text);
```

```
                    diskWindows[device_name]-
>print(errorInfo);
                }
            }else if (request.find("read") !=
std::string::npos) {
                // 读取磁盘
                // request =
"read,blockIndex,byteIndex,length,buffernum"
                int blockIndex = std::stoi(argi(request, 2));
                int byteIndex = std::stoi(argi(request, 3));
                int length = std::stoi(argi(request, 4));
                int buffernum = std::stoi(argi(request, 5));

                int flag;
                if((flag = diskWindows[device_name]-
>readFromFile(blockIndex, byteIndex, length,
deviceQueue.readInBuffer[buffernum])) == SUCCESS){
                    if(logger)
                        std::cout << "设备 " << device_name
<< " 执行进程 " << process_name << " 的任务:[" << request <<
"]\n";

                    // 在窗口中显示任务信息
                    QString taskInfo = "Read Task - Block: "
+ QString::number(blockIndex) +
                                    ", Byte: " +
QString::number(byteIndex) +
                                    ", Length: " +
QString::number(length) +
                                    ", Content: " +
QString::fromStdString(deviceQueue.readInBuffer[buffernum]);
                    diskWindows[device_name]-
>print(taskInfo);
                }else if(flag == OVERSTEP){
                    // 处理越界情况
                    if(logger)
```

```
                                    std::cout << "设备 " << device_name
<< " 执行进程 " << process_name << " 的任务越界:[" << request
<< "]\n";

                                // 在窗口中显示越界信息
                                QString errorInfo = "Error: OverStep -
Block: " + QString::number(blockIndex) +
                                                    ", Byte: " +
QString::number(byteIndex) +
                                                    ", Length: " +
QString::number(length);
                                diskWindows[device_name]-
>print(errorInfo);
                            }else if(flag == FILE_ERR){
                                // 处理文件打开失败情况
                                if(logger)
                                    std::cout << "设备 " << device_name
<< " 执行进程 " << process_name << " 的任务文件打开失败:[" <<
request << "]\n";

                                // 在窗口中显示文件打开失败信息
                                QString errorInfo = "Error: File Open
Failed - Block: " + QString::number(blockIndex) +
                                                    ", Byte: " +
QString::number(byteIndex) +
                                                    ", Length: " +
QString::number(length);
                                diskWindows[device_name]-
>print(errorInfo);
                            }


                        }
                    } // disk

                    // 释放设备
                    deviceQueue.release_device(device_name,
process_name);
                    ++ devicePointer;
```

```
        updateDeviceStatus(); // 更新设备状态显示
    }

    void stop() {
        timer->stop();
    }

    void start() {
        timer->start(10);
    }


private:
    void createDeviceWindows() {
        QPalette palette;
        int x = 0, y = 0;
        for (const auto &device : deviceTable.deviceList) {
            if (device.type == "screen") {
                // 创建屏幕设备窗口
                DeviceWindow *screenWindow = new
DeviceWindow("screen", QString::fromStdString(device.name),
this);
                screenWindow->setStyleSheet("background-
color: rgba(31, 31, 31, 180); color: rgb(230, 230, 230);");
                screenWindow->move(x, y);
                x += 2 * screenWindow->width();
                y += 2 * screenWindow->height();
                screenWindows[device.name] = screenWindow;
                screenWindow->show();
            } else if (device.type == "printer") {
                // 创建打印机设备窗口
                DeviceWindow *printerWindow = new
DeviceWindow("printer", QString::fromStdString(device.name),
this);
                printerWindow->setStyleSheet("background-
color: rgba(31, 31, 31, 180); color: rgb(230, 230, 230);");
                printerWindow->move(x, y);
                x += 2 * printerWindow->width();
                y += 2 * printerWindow->height();
                printerWindows[device.name] = printerWindow;
                printerWindow->show();
            } else if (device.type == "disk") {
                // 创建磁盘设备窗口
                DeviceWindow *diskWindow = new
DeviceWindow("disk", QString::fromStdString(device.name),
this);
                diskWindow->setStyleSheet("background-color:
rgba(31, 31, 31, 180); color: rgb(230, 230, 230);");
                diskWindow->move(x, y);
                x += 2 * diskWindow->width();
                y += 2 * diskWindow->height();
                diskWindows[device.name] = diskWindow;
                diskWindow->show();
            }
        }
    }

    int logger = 0;

    // running遍历使用
    map<string, vector<DevRequest>>::iterator devicePointer;
// 设备指针
    map<string, vector<DevRequest>> occupied_devices; // 临时
设备字典
    int max_priority = 0; // 最高优先级
    map<int, int> running_record; // 运行记录

    QWidget *centralWidget;
    QVBoxLayout *layout;
    QLabel *statusLabel;
    QTimer *timer;
    DeviceTable &deviceTable; // 引用设备信息表
    DeviceQueue &deviceQueue; // 引用设备队列
    QMap<string, DeviceWindow*> screenWindows; // 屏幕设备窗口
    QMap<string, DeviceWindow*> printerWindows; // 打印机设备
窗口
```

```
    QMap<string, DeviceWindow*> diskWindows; // 磁盘设备窗口
};

#endif
```