```cpp
#pragma once

#include <mqueue.h>
#include <unordered_map>
#include <unordered_set>

#include "CommunicationManager.hpp"
#include "MQClientQueueCreationMessage.hpp"
#include "MQClientQueueDeletionMessage.hpp"
#include "MQClientRegistrationMessage.hpp"
#include "Scheduler.hpp"
#include "ShmManager.hpp"

/**
 * @class RegistrationVisitor
 * @brief Handles the registration and management of client message queues.
 *
 * Processes registration messages using Visitor pattern.
 * Registers new clients, opens their message queues, and manages their lifecycle.
 */
class RegistrationVisitor {
  public:
    static RegistrationVisitor& getInstance() {
        static RegistrationVisitor instance;
        return instance;
    };
    void visitRegistration(MQClientRegistrationMessage&);
    void visitQueueCreation(MQClientQueueCreationMessage&);
    void visitQueueDeletion(MQClientQueueDeletionMessage&);
    void dispatch();
    void reg();
    void writeBlocking(int, char*, size_t);

    RegistrationVisitor(const RegistrationVisitor&) = delete;
    RegistrationVisitor& operator=(const RegistrationVisitor&) = delete;

  private:
    RegistrationVisitor();
    int nextClientId = 0;
    std::unordered_set<int> registeredClients;
    std::unordered_map<int, mqd_t> blockingServerToClientQueues, nonBlockingClientToServerQueu
es;
    mqd_t nonBlockingClientToServerQueue, blockingServerToClientQueue;
    Scheduler& scheduler = Scheduler::getInstance();
    CommunicationManager& communicationManager = CommunicationManager::getInstance();
    ShmManager& shmManager = ShmManager::getInstance();
};
```

```
#pragma once

#include "RegistrationVisitor.hpp"

/**
 * @class RegistrationWorker
 * @brief Handles the registration of clients.
 *
 * Spins in infinite loop scanning for registration messages and client request messages.
 * Handles the registration messages and enqueues client request messages for scheduler to dis
patch later.
 */
class RegistrationWorker {
  public:
    RegistrationWorker(){};
    void run();

  private:
    RegistrationVisitor& registrationVisitor = RegistrationVisitor::getInstance();
};
```

```
#pragma once

#include "ISchedulingStrategy.hpp"

/**
 * @class CreditSchedulingStrategy
 * @brief Scheduling strategy that uses credit-based allocation
 *
 * Implements a credit-based scheduling strategy for managing client requests by calling the c
orresponding method on the Scheduler instance.
 * Keeps ownership of all scheduling data inside the Scheduler instance.
 */
class CreditSchedulingStrategy : public ISchedulingStrategy {
  public:
    std::unique_ptr<IShmMessageHandler> schedule(Scheduler&) override;
};
```

```cpp
#pragma once

#include "IShmMessageHandler.hpp"
#include <memory>

class Scheduler;

/**
 * @class ISchedulingStrategy
 * @brief Interface for scheduling strategies
 *
 * Defines the Strategy pattern interface used by the Scheduler to determine the order in whic
h client requests are processed.
 *
 * @param scheduler Reference to the Scheduler instance
 */
class ISchedulingStrategy {
  public:
    virtual std::unique_ptr<IShmMessageHandler> schedule(Scheduler&) = 0;
    virtual ~ISchedulingStrategy() = default;
};
```

```cpp
#pragma once

#include "ISchedulingStrategy.hpp"

/**
 * @class RoundRobinSchedulingStrategy
 * @brief Scheduling strategy that uses round-robin allocation
 *
 * Implements a round-robin scheduling strategy for managing client requests by calling the co
rresponding method on the Scheduler instance.
 * Keeps ownership of all scheduling data inside the Scheduler instance.
 */
class RoundRobinSchedulingStrategy : public ISchedulingStrategy {
  public:
    std::unique_ptr<IShmMessageHandler> schedule(Scheduler&) override;
};
```

```cpp
#pragma once

#include <list>
#include <memory>
#include <mutex>
#include <unordered_map>

#include "CreditSchedulingStrategy.hpp"
#include "ISchedulingStrategy.hpp"
#include "IShmMessageHandler.hpp"
#include "MQClientRequestMessage.hpp"
#include "RoundRobinSchedulingStrategy.hpp"
#include "ShmManager.hpp"

/**
 * @class Scheduler
 * @brief Singleton for managing scheduling of client requests
 *
 * Uses a scheduling strategy (credit, round-robin) to determine the order in which client req
uests are processed.
 * Tracks client registrations and client requests. Scheduling works by maintaining a queue of
 requests for each client and creating a
 * message handler based off the request to process it. This request handler gets passed to a
worker thread that simply calls the process() method.
 *
 * @param numWorkers Number of worker threads available
 * @param schedulingStrategy Scheduling strategy to use
 */
class Scheduler {
  public:
    static void init(size_t numWorkers, std::unique_ptr<ISchedulingStrategy> schedulingStrateg
y) {
        std::call_once(onceFlag,
                       [numWorkers, schedulingStrategy = std::move(schedulingStrategy)]() muta
ble {
                           instance.reset(new Scheduler(numWorkers, std::move(schedulingStrate
gy)));
                       });
    };
    static Scheduler& getInstance() {
        if (!instance) {
            THROW_RUNTIME_ERROR("Scheduler: instance not initialized");
        }
        return *instance;
    };
    Scheduler(const Scheduler&) = delete;
    Scheduler& operator=(const Scheduler&) = delete;

    void registerClient(int, int);
    void deregisterClient(int);
    void enqueue(int, MQClientRequestMessage&);
    std::unique_ptr<IShmMessageHandler> schedule();

  private:
    /**
     * @struct Request
     * @brief Represents a client request.
     *
     * Contains the necessary information to process a client's request.
     *
     * @param clientId The ID of the client making the request.
     * @param requestId The ID of the request.
     * @param fileSize The size of the file being processed.
```

```
     */
    struct Request {
        int clientId;
        int requestId;
        size_t fileSize;

        Request(int clientId, int requestId, size_t fileSize)
            : clientId(clientId), requestId(requestId), fileSize(fileSize){};
    };

    enum class CreditType {
        UNDER,
        OVER,
        IDLE,
    };

    /**
     * @struct Credit
     * @brief Represents credit scheduling state of a client.
     *
     * Keeps track of client's priority, current credit, and type (UNDER, OVER, IDLE).
     * Moves clients between underQueue and overQueue based on their credit status.
     *
     * @param clientId The ID of the client.
     * @param priority The priority of the client.
     * @param credit The initial credit of the client.
     */
    struct Credit {
        int clientId;
        double priority;
        double credit;
        CreditType type;
        std::list<int>::iterator iterator;

        Credit(int clientId, int priority, double credit) : clientId(clientId), priority(prior
ity), credit(credit), type(CreditType::IDLE) {}
        void enqueue(Scheduler& scheduler);
        void dequeue(Scheduler& scheduler);
        Request schedule(Scheduler& scheduler);
    };

    friend struct Credit;

    Scheduler(size_t numWorkers, std::unique_ptr<ISchedulingStrategy> schedulingStrategy)
        : numWorkers(numWorkers), schedulingStrategy(std::move(schedulingStrategy)){};
    static std::unique_ptr<Scheduler> instance;
    static std::once_flag onceFlag;

    size_t numWorkers;
    std::unique_ptr<ISchedulingStrategy> schedulingStrategy;

    int totalCredits = 0;
    std::unordered_map<int, Credit> clientCredits;          // clientId to priority
    std::list<int> underQueue, overQueue;                   // clientIds
    std::unordered_map<int, std::list<Request>> requestMap; // clientId to list of requests
    std::unordered_map<int, std::list<Request>>::iterator
        requestMapIterator; // iterator for requestMap
    size_t requestCount = 0;
    std::mutex requestMapMutex;
    std::condition_variable requestMapCond;
    ShmManager& shmManager = ShmManager::getInstance();

    size_t selectRoundRobinShmSize(Request&);
```

```
    size_t selectCreditShmSize(Request&);
    std::unique_ptr<IShmMessageHandler> createRoundRobinHandler(Request&);
    std::unique_ptr<IShmMessageHandler> createCreditHandler(Request&);
    std::unique_ptr<IShmMessageHandler> scheduleRoundRobin();
    std::unique_ptr<IShmMessageHandler> scheduleCredit();

    friend class RoundRobinSchedulingStrategy;
    friend class CreditSchedulingStrategy;
};
```

```cpp
#pragma once

#include "AbstractShmSegment.hpp"
#include "CommunicationManager.hpp"
#include "IShmMessageHandler.hpp"
#include "RegistrationVisitor.hpp"
#include <memory>

/**
 * @class ServiceFileCompressorHandler
 * @brief Handles the compression of files for a specific client request.
 *
 * Implements the IShmMessageHandler interface to process file compression requests.
 * Manages shared memory segments and semaphores for communication with the client.
 *
 * @param clientId The ID of the client making the request.
 * @param requestId The ID of the request.
 * @param fileSize The size of the file being processed.
 * @param shmSegment The shared memory segment used for communication.
 */
class ServiceFileCompressorHandler : public IShmMessageHandler {
  public:
    ServiceFileCompressorHandler(int clientId, int requestId, size_t fileSize,
                                 std::unique_ptr<AbstractShmSegment> shmSegment)
        : clientId(clientId), requestId(requestId), fileSize(fileSize),
          shmSegment(std::move(shmSegment)), rawData(std::make_unique<char[]>(fileSize)){};
    void process();

  private:
    int clientId;
    int requestId;
    size_t fileSize;
    size_t compressedFileSize;
    size_t fileRecvSize = 0;
    size_t fileSendSize = 0;
    std::unique_ptr<AbstractShmSegment> shmSegment;
    std::unique_ptr<char[]> rawData;
    std::string compressedData;
    CommunicationManager& communicationManager = CommunicationManager::getInstance();
    RegistrationVisitor& registrationVisitor = RegistrationVisitor::getInstance();
    void compressData();
    void processClientData();
    void sendClientData();
    void sendFileData();
};
```

```cpp
#pragma once

#include "Scheduler.hpp"

/**
 * @class ServiceFileCompressorWorker
 * @brief Handles the compression of files.
 *
 * Spins in infinite loop waiting for scheduler to dispatch requests and processes them.
 */
class ServiceFileCompressorWorker {
  public:
    ServiceFileCompressorWorker(int workerId) : workerId(workerId){};
    void run();

  private:
    int workerId;
    Scheduler& scheduler = Scheduler::getInstance();
};
```

```cpp
#include "ArgParse.hpp"
#include "CreditSchedulingStrategy.hpp"
#include "ISchedulingStrategy.hpp"
#include "RegistrationWorker.hpp"
#include "RoundRobinSchedulingStrategy.hpp"
#include "Scheduler.hpp"
#include "ServiceFileCompressorWorker.hpp"
#include "ShmManager.hpp"
#include "Util.hpp"

#include <iostream>
#include <thread>
#include <vector>

static constexpr size_t NUM_WORKERS = 4;

enum class SchedulerType { ROUND_ROBIN, CREDIT };

int main(int argc, char* argv[]) {
    argparse::ArgumentParser program("TinyFile");

    program.add_argument("--n_sms")
        .help("Number of shared memory segments")
        .required()
        .scan<'i', int>();

    program.add_argument("--sms_size")
        .help("Shared memory segment size in bytes")
        .required()
        .scan<'i', int>();

    program.add_argument("--scheduler")
        .help("Set the scheduler algorithm to CREDIT or ROUND_ROBIN")
        .default_value(SchedulerType::ROUND_ROBIN)
        .action([](const std::string& value) {
            if (value == "ROUND_ROBIN") {
                return SchedulerType::ROUND_ROBIN;
            } else if (value == "CREDIT") {
                return SchedulerType::CREDIT;
            } else {
                THROW_RUNTIME_ERROR("Invalid scheduler: " + value);
            }
        });

    int numShmSegments, shmSegmentSize;
    SchedulerType schedulerType;
    try {
        program.parse_args(argc, argv);

        numShmSegments = program.get<int>("--n_sms");
        shmSegmentSize = program.get<int>("--sms_size");
        schedulerType = program.get<SchedulerType>("--scheduler");

        std::cout << "Number of shared memory segments: " << numShmSegments << "\n";
        std::cout << "Shared memory segment size: " << shmSegmentSize << " bytes\n";
    } catch (const std::runtime_error& err) {
        std::cerr << err.what() << std::endl;
        std::cerr << program;
        return 1;
    }

    // Initialize shared memory manager
    ShmManager::init(numShmSegments, shmSegmentSize, 32, true);
```

```
    // Initialize scheduler
    switch (schedulerType) {
    case SchedulerType::ROUND_ROBIN:
        Scheduler::init(NUM_WORKERS, std::make_unique<RoundRobinSchedulingStrategy>());
        break;
    case SchedulerType::CREDIT:
        Scheduler::init(NUM_WORKERS, std::make_unique<CreditSchedulingStrategy>());
        break;
    default:
        THROW_RUNTIME_ERROR("Invalid scheduler");
        break;
    }

    // Start registration worker and file compression workers
    RegistrationWorker registrationWorker;
    std::thread registrationThread(&RegistrationWorker::run, &registrationWorker);
    std::vector<ServiceFileCompressorWorker> fileCompressorWorkers;
    std::vector<std::thread> fileCompressorThreads;
    fileCompressorWorkers.reserve(NUM_WORKERS);
    fileCompressorThreads.reserve(NUM_WORKERS);
    for (size_t i = 0; i < NUM_WORKERS; i++) {
        fileCompressorWorkers.emplace_back(i);
        fileCompressorThreads.emplace_back(&ServiceFileCompressorWorker::run,
                                           &fileCompressorWorkers[i]);
    }
    registrationThread.join();
    for (size_t i = 0; i < NUM_WORKERS; i++) {
        fileCompressorThreads[i].join();
    }
    return 0;
}
```

```cpp
#include "RegistrationVisitor.hpp"

#include "AbstractMQClientRegistrationMessage.hpp"
#include "MQServerIdMessage.hpp"
#include <stdexcept>

/**
 * @brief Constructs a RegistrationVisitor.
 *
 * Initializes global registration queues for client-server + server-client communication.
 */
RegistrationVisitor::RegistrationVisitor() {
    struct mq_attr attr;
    attr.mq_flags = 0;
    attr.mq_maxmsg = REGISTRATION_MAX_MSG;
    attr.mq_msgsize = REGISTRATION_MSG_SIZE;
    attr.mq_curmsgs = 0;
    mq_unlink(communicationManager.getClientToServerRegistrationQueueName().c_str());
    nonBlockingClientToServerQueue =
        mq_open(communicationManager.getClientToServerRegistrationQueueName().c_str(),
                O_CREAT | O_EXCL | O_RDWR | O_NONBLOCK, 0666, &attr);
    if (nonBlockingClientToServerQueue == -1) {
        THROW_RUNTIME_ERROR(
            "RegistrationVisitor: failed to open non-blocking C2S message queue with errno: "
+
            std::to_string(errno));
    }
    mq_unlink(communicationManager.getServerToClientRegistrationQueueName().c_str());
    blockingServerToClientQueue =
        mq_open(communicationManager.getServerToClientRegistrationQueueName().c_str(),
                O_CREAT | O_EXCL | O_RDWR, 0666, &attr);
    if (blockingServerToClientQueue == -1) {
        THROW_RUNTIME_ERROR(
            "RegistrationVisitor: failed to create blocking S2C message queue with errno: " +
            std::to_string(errno));
    }
}

/**
 * @brief Handles the registration of a new client.
 *
 * Assigns a new client ID and sends it back to the client along with shared memory parameters
.
 *
 * @param message The registration message.
 */
void RegistrationVisitor::visitRegistration(MQClientRegistrationMessage& /*message*/) {
    MQServerIdMessage idMessage(nextClientId, shmManager.getNumShmSegments(),
                                shmManager.getShmSegmentSize(), shmManager.getInternalRepSize(
));
    std::unique_ptr<char[]> msg = idMessage.serialize();
    if (mq_send(blockingServerToClientQueue, msg.get(), idMessage.getContentSize(), 0) == -1)
{
        THROW_RUNTIME_ERROR("RegistrationVisitor: failed to send message");
    }
    nextClientId++;
}

/**
 * @brief Handles the creation of a client queue.
 *
 * Adds clientId to the list of registered clients.
 * Adds client created message queues to map.
```

```
 *
 * @param message The queue creation message.
 */
void RegistrationVisitor::visitQueueCreation(MQClientQueueCreationMessage& message) {
    int clientId = message.getClientId();
    if (registeredClients.find(clientId) != registeredClients.end()) {
        THROW_RUNTIME_ERROR("RegistrationVisitor: clientId " + std::to_string(clientId) +
                            " already registered");
    }
    registeredClients.insert(clientId);
    scheduler.registerClient(clientId, message.getPriority());
    blockingServerToClientQueues[clientId] =
        mq_open(communicationManager.getServerToClientRequestQueueName(clientId).c_str(), O_RD
WR);
    if (blockingServerToClientQueues[clientId] == -1) {
        THROW_RUNTIME_ERROR("RegistrationVisitor: failed to open message queue");
    }
    nonBlockingClientToServerQueues[clientId] =
        mq_open(communicationManager.getClientToServerRequestQueueName(clientId).c_str(),
                O_RDWR | O_NONBLOCK);
    if (nonBlockingClientToServerQueues[clientId] == -1) {
        THROW_RUNTIME_ERROR("RegistrationVisitor: failed to open message queue");
    }
}

/**
 * @brief Handles the deletion of a client queue.
 *
 * Removes clientId from the list of registered clients.
 * Closes client created message queues.
 *
 * @param message The queue deletion message.
 */
void RegistrationVisitor::visitQueueDeletion(MQClientQueueDeletionMessage& message) {
    int clientId = message.getClientId();
    if (registeredClients.find(clientId) == registeredClients.end()) {
        THROW_RUNTIME_ERROR("RegistrationVisitor: clientId " + std::to_string(clientId) +
                            " not registered");
    }
    if (mq_close(blockingServerToClientQueues[clientId]) == -1) {
        THROW_RUNTIME_ERROR(
            "RegistrationVisitor: failed to close blockingServerToClientQueue for " +
            std::to_string(clientId) + " with errno: " + std::to_string(errno));
    }
    if (mq_close(nonBlockingClientToServerQueues[clientId]) == -1) {
        THROW_RUNTIME_ERROR(
            "RegistrationVisitor: failed to close nonBlockingClientToServerQueue for " +
            std::to_string(clientId) + " with errno: " + std::to_string(errno));
    }
    registeredClients.erase(clientId);
    scheduler.deregisterClient(clientId);
    blockingServerToClientQueues.erase(clientId);
    nonBlockingClientToServerQueues.erase(clientId);
}

/**
 * @brief Dispatches incoming client requests.
 *
 * Scans every client-to-server queue (non-blocking) for incoming messages and queues them ont
o the scheduler runqueue.
 */
void RegistrationVisitor::dispatch() {
    std::unique_ptr<char[]> buffer = std::make_unique<char[]>(CLIENT_MSG_SIZE);
```

```
    for (auto& [clientId, mq] : nonBlockingClientToServerQueues) {
        while (mq_receive(mq, buffer.get(), CLIENT_MSG_SIZE, nullptr) != -1) {
            MQClientRequestMessage message = MQClientRequestMessage::deserialize(buffer.get())
;
            scheduler.enqueue(clientId, message);
        }
    }
}

/**
 * @brief Registers new clients.
 *
 * Scans global registration queue for incoming client registration messages and processes the
m.
 * Deserializes and demultiplexes messages, then calls accept to visit.
 */
void RegistrationVisitor::reg() {
    std::unique_ptr<char[]> buffer = std::make_unique<char[]>(REGISTRATION_MSG_SIZE);
    while (mq_receive(nonBlockingClientToServerQueue, buffer.get(), REGISTRATION_MSG_SIZE,
                      nullptr) != -1) {
        std::unique_ptr<AbstractMQClientRegistrationMessage> message =
            AbstractMQClientRegistrationMessage::deserialize(buffer.get());
        message->accept(*this);
    }
}

/**
 * @brief Writes a message to server-to-client queue (blocking).
 *
 * @param clientId The ID of the client.
 * @param src The source buffer containing the message.
 * @param size The size of the message.
 */
void RegistrationVisitor::writeBlocking(int clientId, char* src, size_t size) {
    if (size > CLIENT_MSG_SIZE) {
        THROW_RUNTIME_ERROR("RegistrationVisitor: message too large");
    }
    if (registeredClients.find(clientId) == registeredClients.end()) {
        THROW_RUNTIME_ERROR("RegistrationVisitor: clientId " + std::to_string(clientId) +
                            " not registered");
    }
    if (mq_send(blockingServerToClientQueues[clientId], src, size, 0) == -1) {
        THROW_RUNTIME_ERROR("RegistrationVisitor: failed to send message");
    }
}
```

```cpp
#include "RegistrationWorker.hpp"

/**
 * @brief Runs the registration worker.
 *
 * Continuously registers clients and dispatches their requests.
 */
void RegistrationWorker::run() {
    while (true) {
        registrationVisitor.reg();
        registrationVisitor.dispatch();
    }
}
```

```cpp
#include "CreditSchedulingStrategy.hpp"

#include "Scheduler.hpp"

/**
 * @brief Delegates to corresponding scheduling method in Scheduler
 *
 * @param scheduler Reference to the Scheduler instance
 */
std::unique_ptr<IShmMessageHandler> CreditSchedulingStrategy::schedule(Scheduler& scheduler) {
    return scheduler.scheduleCredit();
}
```

```cpp
#include "RoundRobinSchedulingStrategy.hpp"

#include "Scheduler.hpp"

/**
 * @brief Delegates to corresponding scheduling method in Scheduler
 *
 * @param scheduler Reference to the Scheduler instance
 */
std::unique_ptr<IShmMessageHandler> RoundRobinSchedulingStrategy::schedule(Scheduler& schedule
r) {
    return scheduler.scheduleRoundRobin();
}
```

```cpp
#include "Scheduler.hpp"
#include "ServiceFileCompressorHandler.hpp"
#include "CommunicationManager.hpp"
#include "Util.hpp"

/**
 * @brief Enqueues client into appropriate credit queue
 *
 * If client still has positive credit, it is enqueued in the underQueue.
 * If client has no credit, it is enqueued in the overQueue.
 *
 * @param scheduler Reference to the Scheduler instance
 */
void Scheduler::Credit::enqueue(Scheduler& scheduler) {
    if (type == CreditType::IDLE) {
        if (credit > 0) {
            type = CreditType::UNDER;
            scheduler.underQueue.push_back(clientId);
            iterator = std::prev(scheduler.underQueue.end());
        } else {
            type = CreditType::OVER;
            scheduler.overQueue.push_back(clientId);
            iterator = std::prev(scheduler.overQueue.end());
        }
    }
}

/**
 * @brief Dequeues client from its current credit queue
 *
 * @param scheduler Reference to the Scheduler instance
 */
void Scheduler::Credit::dequeue(Scheduler& scheduler) {
    switch (type) {
    case CreditType::UNDER:
        scheduler.underQueue.erase(iterator);
        break;
    case CreditType::OVER:
        scheduler.overQueue.erase(iterator);
        break;
    default:
        THROW_RUNTIME_ERROR("Scheduler: schedule called on idle Credit");
        break;
    }
}

/**
 * @brief Schedules a client request.
 *
 * Selects first request from client's request queue and schedules it.
 * Updates all clients' credits based on file size and priority.
 *
 * @return The scheduled client request.
 */
Scheduler::Request Scheduler::Credit::schedule(Scheduler& scheduler) {
    if (scheduler.requestMap[clientId].empty()) {
        THROW_RUNTIME_ERROR("Scheduler: schedule called on Credit with empty runqueue");
    }
    Request request = scheduler.requestMap[clientId].front();
    scheduler.requestMap[clientId].pop_front();
    dequeue(scheduler);
    for (auto& [otherClientId, credit] : scheduler.clientCredits) {
        if (otherClientId == clientId) {
```

```
            continue;
        }
        credit.credit += credit.priority * request.fileSize / DEFAULT_PRIORITY /
                         FILE_CREDIT_MULTIPLIER / scheduler.numWorkers;
        if (credit.type != CreditType::IDLE) {
            credit.dequeue(scheduler);
            credit.type = CreditType::IDLE;
            credit.enqueue(scheduler);
        }
    }
    credit -= request.fileSize / FILE_CREDIT_MULTIPLIER / scheduler.numWorkers;
    if (!scheduler.requestMap[clientId].empty()) {
        if (credit > 0) {
            type = CreditType::UNDER;
            scheduler.underQueue.push_back(clientId);
            iterator = std::prev(scheduler.underQueue.end());
        } else {
            type = CreditType::OVER;
            scheduler.overQueue.push_back(clientId);
            iterator = std::prev(scheduler.overQueue.end());
        }
    } else {
        type = CreditType::IDLE;
    }
    return request;
}

std::unique_ptr<Scheduler> Scheduler::instance;
std::once_flag Scheduler::onceFlag;

/**
 * @brief Registers a new client.
 *
 * Adds client with corresponding priority/credits to the scheduler.
 *
 * @param clientId The ID of the client to register.
 * @param priority The priority of the client.
 */
void Scheduler::registerClient(int clientId, int priority) {
    std::lock_guard<std::mutex> lock(requestMapMutex);
    clientCredits.emplace(clientId, Credit(clientId, priority, priority));
    totalCredits += priority;
}

/**
 * @brief Deregisters an existing client.
 *
 * Removes client from the scheduler.
 *
 * @param clientId The ID of the client to deregister.
 */
void Scheduler::deregisterClient(int clientId) {
    std::lock_guard<std::mutex> lock(requestMapMutex);
    auto it = clientCredits.find(clientId);
    if (it == clientCredits.end()) {
        THROW_RUNTIME_ERROR("Scheduler: clientId " + std::to_string(clientId) + " not register
ed");
    }
    totalCredits -= it->second.priority;
    clientCredits.erase(it);
    requestMap.erase(clientId);
}
```

```
/**
 * @brief Enqueues a client request.
 *
 * Adds a new client request to the client's request queue.
 * Wakes up workers waiting for new requests.
 *
 * @param clientId The ID of the client making the request.
 * @param MQClientRequestMessage The client request message.
 */
void Scheduler::enqueue(int clientId, MQClientRequestMessage& MQClientRequestMessage) {
    std::lock_guard<std::mutex> lock(requestMapMutex);
    requestMap[clientId].emplace_back(clientId, MQClientRequestMessage.getRequestId(),
                                      MQClientRequestMessage.getFileSize());
    requestCount++;
    auto it = clientCredits.find(clientId);
    if (it == clientCredits.end()) {
        THROW_RUNTIME_ERROR("Scheduler: clientId " + std::to_string(clientId) + " not register
ed");
    }
    it->second.enqueue(*this);
    requestMapCond.notify_one();
}

/**
 * @brief Schedules a client request.
 *
 * @return A unique pointer to the scheduled message handler.
 */
std::unique_ptr<IShmMessageHandler> Scheduler::schedule() {
    return schedulingStrategy->schedule(*this);
}

/**
 * @brief Schedules a client request using the round-robin strategy.
 *
 * Scans all client request queues in order to find the next request to process.
 * Stores requestMapIterator so next call can resume from the last processed request.
 *
 * @return A unique pointer to the scheduled message handler.
 */
std::unique_ptr<IShmMessageHandler> Scheduler::scheduleRoundRobin() {
    std::unique_lock<std::mutex> lock(requestMapMutex);
    requestMapCond.wait(lock, [this] { return requestCount; });

    while (requestMapIterator != requestMap.end()) {
        if (!requestMapIterator->second.empty()) {
            Request request = requestMapIterator->second.front();
            requestMapIterator->second.pop_front();
            requestMapIterator++;
            requestCount--;
            return createRoundRobinHandler(request);
        }
        requestMapIterator++;
    }
    requestMapIterator = requestMap.begin();
    while (requestMapIterator != requestMap.end()) {
        if (!requestMapIterator->second.empty()) {
            Request request = requestMapIterator->second.front();
            requestMapIterator->second.pop_front();
            requestMapIterator++;
            requestCount--;
            return createRoundRobinHandler(request);
        }
```

```
        requestMapIterator++;
    }
    THROW_RUNTIME_ERROR("FileCompressionHandler: no requests to process");
}

/**
 * @brief Schedules a client request using the credit strategy.
 *
 * Checks top of underQueue first (clients who still have credits).
 * If no clients in underQueue, checks overQueue (clients who consumed all their credits).
 *
 * @return A unique pointer to the scheduled message handler.
 */
std::unique_ptr<IShmMessageHandler> Scheduler::scheduleCredit() {
    std::unique_lock<std::mutex> lock(requestMapMutex);
    requestMapCond.wait(lock, [this] { return requestCount; });

    if (!underQueue.empty()) {
        int clientId = underQueue.front();
        auto it = clientCredits.find(clientId);
        if (it == clientCredits.end()) {
            THROW_RUNTIME_ERROR("Scheduler: clientId " + std::to_string(clientId) +
                                " not registered");
        }
        Request request = it->second.schedule(*this);
        requestCount--;
        std::unique_ptr<IShmMessageHandler> handler = createCreditHandler(request);
        return handler;
    } else {
        int clientId = overQueue.front();
        auto it = clientCredits.find(clientId);
        if (it == clientCredits.end()) {
            THROW_RUNTIME_ERROR("Scheduler: clientId " + std::to_string(clientId) +
                                " not registered");
        }
        Request request = it->second.schedule(*this);
        requestCount--;
        std::unique_ptr<IShmMessageHandler> handler = createCreditHandler(request);
        return handler;
    }
}

/**
 * @brief Creates a message handler for a round-robin client request.
 *
 * Selects the appropriate shared memory segment and creates the handler.
 *
 * @param request The client request to process.
 * @return A unique pointer to the created message handler.
 */
std::unique_ptr<IShmMessageHandler> Scheduler::createRoundRobinHandler(Request& request) {
    // Assumes requestMapMutex is held
    return std::make_unique<ServiceFileCompressorHandler>(
        request.clientId, request.requestId, request.fileSize,
        shmManager.getShmSegment(selectRoundRobinShmSize(request)));
}

/**
 * @brief Creates a message handler for a credit client request.
 *
 * Selects the appropriate shared memory segment and creates the handler.
 *
 * @param request The client request to process.
```

```
 * @return A unique pointer to the created message handler.
 */
std::unique_ptr<IShmMessageHandler> Scheduler::createCreditHandler(Request& request) {
    // Assumes requestMapMutex is held
    return std::make_unique<ServiceFileCompressorHandler>(
        request.clientId, request.requestId, request.fileSize,
        shmManager.getShmSegment(selectCreditShmSize(request)));
}

/**
 * @brief Selects the shared memory size for a round-robin client request.
 *
 * @param request The client request to process.
 * @return The selected shared memory size.
 */
size_t Scheduler::selectRoundRobinShmSize(Request& request) {
    // Assumes requestMapMutex is held
    return std::max(shmManager.getInternalRepSize(),
                    std::min((shmManager.getSize() / (requestCount + numWorkers) +
                              shmManager.getInternalRepSize() - 1) /
                                 shmManager.getInternalRepSize() * shmManager.getInternalRepSi
ze(),
                             (request.fileSize + shmManager.getInternalRepSize() - 1) /
                                 shmManager.getInternalRepSize() *
                                 shmManager.getInternalRepSize()));
}

/**
 * @brief Selects the shared memory size for a credit client request.
 *
 * @param request The client request to process.
 * @return The selected shared memory size.
 */
size_t Scheduler::selectCreditShmSize(Request& request) {
    // Assumes requestMapMutex is held
    return std::max(shmManager.getInternalRepSize(),
                    std::min((shmManager.getSize() / (requestCount + numWorkers) +
                              shmManager.getInternalRepSize() - 1) /
                                 shmManager.getInternalRepSize() * shmManager.getInternalRepSi
ze(),
                             (request.fileSize + shmManager.getInternalRepSize() - 1) /
                                 shmManager.getInternalRepSize() *
                                 shmManager.getInternalRepSize()));
}
```

```cpp
#include "ServiceFileCompressorHandler.hpp"

#include "MQServerRequestMessage.hpp"
#include "ShmClientDataMessage.hpp"
#include "ShmServerDataMessage.hpp"
#include "ShmServerFileMessage.hpp"
#include "Util.hpp"
#include "snappy.h"

#include <cassert>
#include <chrono>
#include <fcntl.h>
#include <semaphore.h>
#include <stdexcept>
#include <sys/stat.h>
#include <thread>

/**
 * @brief Processes the file compression request.
 *
 * Handles the entire lifecycle of a file compression request,
 * from receiving the request to sending the response back to the client.
 */
void ServiceFileCompressorHandler::process() {
    // reset semaphores
    if (sem_unlink(communicationManager.getShmClientSemaphoreName(clientId, requestId).c_str()
) ==
            -1 &&
        errno != ENOENT) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to unlink shmClientSemaphore
");
    }
    sem_t* shmClientSemaphore =
        sem_open(communicationManager.getShmClientSemaphoreName(clientId, requestId).c_str(),
                 O_CREAT | O_EXCL | O_RDWR, 0666, 0);
    if (shmClientSemaphore == SEM_FAILED) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to open shmClientSemaphore")
;
    }
    if (sem_unlink(communicationManager.getShmServerSemaphoreName(clientId, requestId).c_str()
) ==
            -1 &&
        errno != ENOENT) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to unlink shmServerSemaphore
");
    }
    sem_t* shmServerSemaphore =
        sem_open(communicationManager.getShmServerSemaphoreName(clientId, requestId).c_str(),
                 O_CREAT | O_EXCL | O_RDWR, 0666, 1);
    if (shmServerSemaphore == SEM_FAILED) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to open shmServerSemaphore")
;
    }
    if (sem_unlink(communicationManager.getShmSenseSemaphoreName(clientId, requestId).c_str())
 ==
            -1 &&
        errno != ENOENT) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to unlink shmSenseSemaphore"
);
    }
    sem_t* shmSenseSemaphore =
        sem_open(communicationManager.getShmSenseSemaphoreName(clientId, requestId).c_str(),
                 O_CREAT | O_EXCL | O_RDWR, 0666, 0);
```

```
    if (shmSenseSemaphore == SEM_FAILED) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to open shmSenseSemaphore");
    }

    // write message to client that signals all semaphores are ready
    MQServerRequestMessage requestMessage(requestId, shmSegment.get());
    std::unique_ptr<char[]> serializedRequestMessage = requestMessage.serialize();
    registrationVisitor.writeBlocking(clientId, serializedRequestMessage.get(),
                                      requestMessage.getContentSize());

    /**
     * communication protocol:
     * 1. client writes data to shared memory segment and notifies server by posting on shmCli
entSemaphore
     * 2. server processes the request by copying file contents into its own buffer
     * 3. server sends the response back to the client by posting on shmServerSemaphore
     */
    while (fileRecvSize < fileSize) {
        // wait until client data is available
        if (sem_wait(shmClientSemaphore) == -1) {
            THROW_RUNTIME_ERROR(
                "ServiceFileCompressorHandler: failed to wait on shmClientSemaphore");
        }
        // process client data
        processClientData();
        // signal that server has processed client data
        if (sem_post(shmServerSemaphore) == -1) {
            THROW_RUNTIME_ERROR(
                "ServiceFileCompressorHandler: failed to post on shmServerSemaphore");
        }
    }
    // compress data
    compressData();
    // used to decrement count to 0 so next round of transfer can proceed
    if (sem_wait(shmServerSemaphore) == -1) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to wait on shmServerSemaphor
e");
    }
    // send post-compression file metadata
    sendFileData();
    // signal that server has sent file metadata
    if (sem_post(shmSenseSemaphore) == -1) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to post on shmSenseSemaphore
");
    }
    while (fileSendSize < compressedFileSize) {
        // wait for client to be ready to receive data
        if (sem_wait(shmServerSemaphore) == -1) {
            THROW_RUNTIME_ERROR(
                "ServiceFileCompressorHandler: failed to wait on shmServerSemaphore");
        }
        // send data
        sendClientData();
        // signal that server has sent client data
        if (sem_post(shmClientSemaphore) == -1) {
            THROW_RUNTIME_ERROR(
                "ServiceFileCompressorHandler: failed to post on shmClientSemaphore");
        }
    }

    // cleanup
    if (sem_close(shmClientSemaphore) == -1) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to close shmClientSemaphore"
```

```
);
    }
    if (sem_close(shmServerSemaphore) == -1) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to close shmServerSemaphore"
);
    }
    if (sem_close(shmSenseSemaphore) == -1) {
        THROW_RUNTIME_ERROR("ServiceFileCompressorHandler: failed to close shmSenseSemaphore")
;
    }
}

/**
 * @brief Processes the client data received via the shared memory segment.
 *
 * Extracts client data from the shared memory segment and copies it to the
 * rawData buffer.
 */
void ServiceFileCompressorHandler::processClientData() {
    ShmClientDataMessage msg(shmSegment.get());
    msg.parse();
    msg.copyData(rawData.get() + fileRecvSize, fileSize - fileRecvSize);
    fileRecvSize += msg.getDataSize();
}

/**
 * @brief Sends data to the client via the shared memory segment.
 *
 * Writes data from the compressedData buffer to the shared memory segment.
 */
void ServiceFileCompressorHandler::sendClientData() {
    ShmServerDataMessage msg(shmSegment.get(), compressedData.data() + fileSendSize,
                             std::min(compressedFileSize - fileSendSize,
                                      shmSegment->getSize() - ShmServerDataMessage::headerSize
()));
    msg.write();
    fileSendSize += msg.getDataSize();
}

/**
 * @brief Sends the post-compression file metadata to the client.
 */
void ServiceFileCompressorHandler::sendFileData() {
    ShmServerFileMessage msg(shmSegment.get(), compressedFileSize);
    msg.write();
}

/**
 * @brief Compresses the client data using Snappy compression.
 */
void ServiceFileCompressorHandler::compressData() {
    snappy::Compress(rawData.get(), fileSize, &compressedData);
    compressedFileSize = compressedData.size();
}
```

```cpp
#include "ServiceFileCompressorWorker.hpp"

/**
 * @brief Runs the service file compressor worker.
 *
 * Continuously asks scheduler for next request and processes it.
 */
void ServiceFileCompressorWorker::run() {
    while (true) {
        std::unique_ptr<IShmMessageHandler> handler = scheduler.schedule();
        handler->process();
    }
}
```

```cpp
#include <gtest/gtest.h>

#include "MQClientQueueCreationMessage.hpp"

TEST(MQClientQueueCreationMessage, full) {
    MQClientQueueCreationMessage sendMessage(1, 2);
    std::unique_ptr<char[]> buffer = sendMessage.serialize();
    MQClientQueueCreationMessage recvMessage =
        MQClientQueueCreationMessage::deserialize(buffer.get());
    EXPECT_EQ(recvMessage.getClientId(), 1);
    EXPECT_EQ(recvMessage.getPriority(), 2);

    MQClientQueueCreationMessage sendMessage2(1);
    std::unique_ptr<char[]> buffer2 = sendMessage2.serialize();
    MQClientQueueCreationMessage recvMessage2 =
        MQClientQueueCreationMessage::deserialize(buffer2.get());
    EXPECT_EQ(recvMessage2.getClientId(), 1);
    EXPECT_EQ(recvMessage2.getPriority(), 64);
}
```

```cpp
#include <gtest/gtest.h>

#include "AbstractShmSegment.hpp"
#include "ShmClientDataMessage.hpp"
#include "ShmManager.hpp"
#include "ShmServerDataMessage.hpp"
#include "ShmServerFileMessage.hpp"

TEST(ShmServerDataMessage, full) {
    ShmManager::init(10, 1024, 32, true);
    ShmManager& manager = ShmManager::getInstance();
    std::string msg = "Hello, World!";
    std::unique_ptr<AbstractShmSegment> segment = manager.getShmSegment(64);
    ShmServerDataMessage sendMessage(segment.get(), msg.data(), msg.size());
    sendMessage.write();
    ShmServerDataMessage recvMessage(segment.get());
    recvMessage.parse();
    std::unique_ptr<char[]> recvBuffer = std::make_unique<char[]>(msg.size());
    recvMessage.copyData(recvBuffer.get(), msg.size());
    EXPECT_EQ(std::string(recvBuffer.get()), msg);
}

TEST(ShmClientDataMessage, full) {
    ShmManager::init(10, 1024, 32, true);
    ShmManager& manager = ShmManager::getInstance();
    std::string msg = "Hello, World!";
    std::unique_ptr<AbstractShmSegment> segment = manager.getShmSegment(64);
    ShmClientDataMessage sendMessage(segment.get(), msg.data(), msg.size());
    sendMessage.write();
    ShmClientDataMessage recvMessage(segment.get());
    recvMessage.parse();
    std::unique_ptr<char[]> recvBuffer = std::make_unique<char[]>(msg.size());
    recvMessage.copyData(recvBuffer.get(), msg.size());
    EXPECT_EQ(std::string(recvBuffer.get()), msg);
}

TEST(ShmServerFileMessage, full) {
    ShmManager::init(10, 1024, 32, true);
    ShmManager& manager = ShmManager::getInstance();
    std::unique_ptr<AbstractShmSegment> segment = manager.getShmSegment(64);

    ShmServerFileMessage sendMessage(segment.get(), 42);
    sendMessage.write();
    ShmServerFileMessage recvMessage(segment.get());
    recvMessage.parse();
    EXPECT_EQ(recvMessage.getFileSize(), 42);

    sendMessage = ShmServerFileMessage(segment.get(), 100);
    sendMessage.write();
    recvMessage = ShmServerFileMessage(segment.get());
    recvMessage.parse();
    EXPECT_EQ(recvMessage.getFileSize(), 100);

    sendMessage = ShmServerFileMessage(segment.get(), 1000000000);
    sendMessage.write();
    recvMessage = ShmServerFileMessage(segment.get());
    recvMessage.parse();
    EXPECT_EQ(recvMessage.getFileSize(), 1000000000);
}
```

```
#include <gtest/gtest.h>

#include "AbstractShmSegment.hpp"
#include "ShmManager.hpp"

TEST(ShmManagerTest, initialize) {
    ShmManager::init(10, 1024, 32, true);
    ShmManager& manager = ShmManager::getInstance();
    EXPECT_EQ(manager.getSize(), 10240);
    EXPECT_EQ(manager.getAvailableSize(), 10240);
    EXPECT_EQ(manager.numSegments, 10);
    EXPECT_EQ(manager.numInternalSegments, 320);
    EXPECT_EQ(manager.internalRepSize, 32);
    std::unique_ptr<AbstractShmSegment> segment = manager.getShmSegment(64);
    EXPECT_EQ(manager.getAvailableSize(), 10176);
    for (int i = 0; i < 2; i++) {
        EXPECT_EQ(manager.availableSegments[i], false);
    }
    std::unique_ptr<AbstractShmSegment> segment2 = manager.getShmSegment(64);
    EXPECT_EQ(manager.getAvailableSize(), 10112);
    for (int i = 2; i < 4; i++) {
        EXPECT_EQ(manager.availableSegments[i], false);
    }
    std::unique_ptr<AbstractShmSegment> segment3 = manager.getShmSegment(1024);
    EXPECT_EQ(manager.getAvailableSize(), 9088);
    for (int i = 4; i < 36; i++) {
        EXPECT_EQ(manager.availableSegments[i], false);
    }
    for (int i = 36; i < 320; i++) {
        EXPECT_EQ(manager.availableSegments[i], true);
    }
    EXPECT_THROW(manager.getShmSegment(33), std::runtime_error);
    segment.reset();
    segment2.reset();
    segment3.reset();
    EXPECT_EQ(manager.getAvailableSize(), 10240);
    for (int i = 0; i < 320; i++) {
        EXPECT_EQ(manager.availableSegments[i], true);
    }
}

TEST(ShmSegmentTest, full) {
    ShmManager::init(10, 1024, 32, true);
    ShmManager& manager = ShmManager::getInstance();
    // ContiguousShmSegment
    std::unique_ptr<AbstractShmSegment> contiguousSegment = manager.getShmSegment(32);
    contiguousSegment->write("Hello, World!", 14);
    std::unique_ptr<char[]> contiguousShmBuffer = std::make_unique<char[]>(18);
    contiguousSegment->read(contiguousShmBuffer.get(), 14);
    EXPECT_EQ(std::string(contiguousShmBuffer.get()), "Hello, World!");
    contiguousSegment->write("Hello, World!", 14, 4);
    contiguousSegment->read(contiguousShmBuffer.get(), 14, 4);
    EXPECT_EQ(std::string(contiguousShmBuffer.get()), "Hello, World!");
    contiguousSegment->read(contiguousShmBuffer.get(), 18);
    EXPECT_EQ(std::string(contiguousShmBuffer.get()), "HellHello, World!");
    // FragmentedShmSegment
    std::unique_ptr<AbstractShmSegment> fragmentedSegment = manager.getShmSegment(64);
    fragmentedSegment->write("Really long message that goes past the 32 byte boundary", 56);
    std::unique_ptr<char[]> fragmentedShmBuffer = std::make_unique<char[]>(60);
    fragmentedSegment->read(fragmentedShmBuffer.get(), 56);
    EXPECT_EQ(std::string(fragmentedShmBuffer.get()),
              "Really long message that goes past the 32 byte boundary");
    fragmentedSegment->write("Really long message that goes past the 32 byte boundary", 56, 4)
```

```
;
    fragmentedSegment->read(fragmentedShmBuffer.get(), 56, 4);
    EXPECT_EQ(std::string(fragmentedShmBuffer.get()),
              "Really long message that goes past the 32 byte boundary");
    fragmentedSegment->read(fragmentedShmBuffer.get(), 60);
    EXPECT_EQ(std::string(fragmentedShmBuffer.get()),
              "RealReally long message that goes past the 32 byte boundary");
    contiguousSegment.reset();
    fragmentedSegment.reset();
}
```