```cpp
#pragma once

#include <algorithm>
#include <cstdint>
#include <exception>
#include <tuple>
#include <vector>

#include "exceptions.h"
#include "types.h"

namespace MemoryManagement {
inline MemoryState allocateMemory(const MemoryState &state,
                                  uint32_t blockIndex,
                                  int32_t pid,
                                  int32_t pages) {
  auto [blocks, freeBlocks] = state;

  auto block = blocks.at(blockIndex);
  if (block.pid() != -1) {
    throw OperationException("BLOCK_IS_USED");
  } else if (block.size() < pages) {
    throw OperationException("TOO_SMALL");
  }

  auto allocatedBlock = MemoryBlock(pid, block.address(), pages);
  auto freeBlockSize = block.size() - pages;
  auto freeBlockAddress = block.address() + pages;

  blocks.erase(blocks.begin() + blockIndex);
  if (freeBlockSize > 0) {
    blocks.insert(blocks.begin() + blockIndex,
                  MemoryBlock(-1, freeBlockAddress, freeBlockSize));
  }
  blocks.insert(blocks.begin() + blockIndex, allocatedBlock);

  auto pos = std::find(freeBlocks.begin(), freeBlocks.end(), block);
  freeBlocks.erase(pos);
  if (freeBlockSize > 0) {
    freeBlocks.emplace_back(-1, freeBlockAddress, freeBlockSize);
  }

  return {blocks, freeBlocks};
}

inline MemoryState
freeMemory(const MemoryState &state, int32_t pid, uint32_t blockIndex) {
  auto [blocks, freeBlocks] = state;

  auto block = blocks.at(blockIndex);
  if (block.pid() != pid) {
    throw OperationException("PID_MISMATCH");
  }

  blocks[blockIndex] = MemoryBlock(-1, block.address(), block.size());
  freeBlocks.push_back(blocks[blockIndex]);

  return {blocks, freeBlocks};
}
```

```cpp
inline MemoryState defragmentMemory(const MemoryState &state) {
  auto [blocks, freeBlocks] = state;

  int32_t address = 0;
  int32_t freeMemory = 0;
  std::vector<MemoryBlock> newBlocks;

  for (const auto &block : blocks) {
    if (block.pid() != -1) {
      newBlocks.emplace_back(block.pid(), address, block.size());
      address += block.size();
    } else {
      freeMemory += block.size();
    }
  }

  newBlocks.emplace_back(-1, address, freeMemory);
  freeBlocks = {{-1, address, freeMemory}};

  return {newBlocks, freeBlocks};
}

inline MemoryState compressMemory(const MemoryState &state,
                                  uint32_t startBlockIndex) {
  auto [blocks, freeBlocks] = state;

  std::vector<MemoryBlock> newBlocks(blocks.begin(),
                                     blocks.begin() + startBlockIndex);

  uint32_t currentBlock = startBlockIndex;
  uint32_t compressingBlocks = 0;
  int32_t address = blocks[startBlockIndex].address();
  int32_t freeMemory = 0;
  while (currentBlock < blocks.size() && blocks[currentBlock].pid() == -1) {
    freeMemory += blocks[currentBlock].size();
    auto pos =
        std::find(freeBlocks.begin(), freeBlocks.end(), blocks[currentBlock]);
    freeBlocks.erase(pos);
    currentBlock += 1;
    compressingBlocks += 1;
  }

  if (compressingBlocks < 2) {
    throw OperationException("SINGLE_BLOCK");
  }

  newBlocks.emplace_back(-1, address, freeMemory);
  newBlocks.insert(
      newBlocks.end(), blocks.begin() + currentBlock, blocks.end());
  freeBlocks.emplace_back(-1, address, freeMemory);

  return {newBlocks, freeBlocks};
}
} // namespace MemoryManagement

#pragma once

#include <cstdint>
```

```cpp
#include <memory>
#include <string>

#include <mapbox/variant.hpp>
#include <nlohmann/json.hpp>

#include "exceptions.h"

namespace MemoryManagement {

class CreateProcessReq {
private:
  int32_t _pid;

  int32_t _bytes;

public:
  CreateProcessReq &operator=(const CreateProcessReq &rhs) = default;

  int32_t pid() const { return _pid; }

  int32_t bytes() const { return _bytes; }

  int32_t pages() const {
    return _bytes % 4096 == 0 ? _bytes / 4096 : (_bytes + 4096) / 4096;
  }

  nlohmann::json dump() const {
    return {{"type", "CREATE_PROCESS"}, {"pid", _pid}, {"bytes", _bytes}};
  }

  CreateProcessReq(int32_t pid, int32_t bytes) : _pid(pid), _bytes(bytes) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
    if (bytes < 1 || bytes > 256 * 4096) {
      throw RequestException("INVALID_BYTES");
    }
  }
};

class TerminateProcessReq {
private:
  int32_t _pid;

public:
  int32_t pid() const { return _pid; }

  TerminateProcessReq &operator=(const TerminateProcessReq &rhs) = default;

  nlohmann::json dump() const {
    return {{"type", "TERMINATE_PROCESS"}, {"pid", _pid}};
  }

  TerminateProcessReq(int32_t pid) : _pid(pid) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
  }
```

```
};

class AllocateMemory {
private:
  int32_t _pid;

  int32_t _bytes;

public:
  int32_t pid() const { return _pid; }

  int32_t bytes() const { return _bytes; }

  int32_t pages() const {
    return _bytes % 4096 == 0 ? _bytes / 4096 : (_bytes + 4096) / 4096;
  }

  AllocateMemory &operator=(const AllocateMemory &rhs) = default;

  nlohmann::json dump() const {
    return {{"type", "ALLOCATE_MEMORY"}, {"pid", _pid}, {"bytes", _bytes}};
  }

  AllocateMemory(int32_t pid, int32_t bytes) : _pid(pid), _bytes(bytes) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
    if (bytes < 1 || bytes > 256 * 4096) {
      throw RequestException("INVALID_BYTES");
    }
  }
};

class FreeMemory {
private:
  int32_t _pid;

  int32_t _address;

public:
  int32_t pid() const { return _pid; }

  int32_t address() const { return _address; }

  FreeMemory &operator=(const FreeMemory &rhs) = default;

  nlohmann::json dump() const {
    return {{"type", "FREE_MEMORY"}, {"pid", _pid}, {"address", _address}};
  }

  FreeMemory(int32_t pid, int32_t address) : _pid(pid), _address(address) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
    if (address < 0 || address > 255) {
      throw RequestException("INVALID_ADDRESS");
    }
  }
};
```

```
using Request = mapbox::util::
    variant<CreateProcessReq, TerminateProcessReq, AllocateMemory, FreeMemory>;
} // namespace MemoryManagement

#pragma once

#include <algorithm>
#include <cstdint>
#include <exception>
#include <memory>
#include <string>

#include <mapbox/variant.hpp>

#include "operations.h"
#include "requests.h"
#include "types.h"

namespace MemoryManagement {
using std::shared_ptr;

enum class StrategyType {
  FIRST_APPROPRIATE,
  MOST_APPROPRIATE,
  LEAST_APPROPRIATE
};

class AbstractStrategy {
public:
  virtual ~AbstractStrategy() = default;

  AbstractStrategy(StrategyType type) : type(type) {}

  const StrategyType type;

  virtual std::string toString() const = 0;

  MemoryState processRequest(const Request &request,
                             const MemoryState &state) const {
    return request.match([this, state](const auto &req) {
      return this->processRequest(req, state);
    });
  }

  MemoryState processRequest(const CreateProcessReq &request,
                             const MemoryState &state) const {
    return sortFreeBlocks(allocateMemoryGeneral(
        AllocateMemory(request.pid(), request.bytes()), state, true));
  }

  MemoryState processRequest(const TerminateProcessReq &request,
                             const MemoryState &state) const {
    auto currentState = state;

    while (true) {
      auto [blocks, freeBlocks] = currentState;
      // ищем очередной блок памяти, выделенный процессу
      auto pos = std::find_if(
```

```
        blocks.begin(), blocks.end(), [&request](const auto &block) {
          return request.pid() == block.pid();
        });
      if (pos == blocks.end()) {
        break;
      }

      // освобождаем блок
      uint32_t index = static_cast<uint32_t>(pos - blocks.begin());
      currentState = freeMemory(currentState, request.pid(), index);
    }

    // сжимаем память
    // сортируем свободные блоки
    return sortFreeBlocks(compressAllMemory(currentState));
  }

  MemoryState processRequest(const AllocateMemory &request,
                             const MemoryState &state) const {
    return sortFreeBlocks(allocateMemoryGeneral(request, state, false));
  }

  MemoryState processRequest(const FreeMemory &request,
                             const MemoryState &state) const {
    auto currentState = state;
    auto [blocks, freeBlocks] = currentState;

    // ищем блок, начинающийся с заданного адреса
    auto pos = std::find_if(
        blocks.begin(), blocks.end(), [&request](const auto &block) {
          return block.address() == request.address();
        });
    // если такого блока нет, игнорируем заявку
    if (pos == blocks.end()) {
      return state;
    }
    // если блок выделен другому процессу, игнорируем заявку
    if (pos->pid() != request.pid()) {
      return state;
    }

    // освобождаем блок
    uint32_t index = static_cast<uint32_t>(pos - blocks.begin());
    currentState = freeMemory(state, request.pid(), index);

    // сжимаем память
    // сортируем свободные блоки
    return sortFreeBlocks(compressAllMemory(currentState));
  }

protected:
  virtual MemoryState sortFreeBlocks(const MemoryState &state) const = 0;

  virtual std::vector<MemoryBlock>::const_iterator
  findFreeBlock(const std::vector<MemoryBlock> &blocks,
                const std::vector<MemoryBlock> &freeBlocks,
                int32_t size) const final {
    auto freeBlockPos = std::find_if(
        freeBlocks.cbegin(), freeBlocks.cend(), [&size](const auto &block) {
```

```
        return size <= block.size();
      });

  if (freeBlockPos != freeBlocks.cend()) {
    return std::find(blocks.cbegin(), blocks.cend(), *freeBlockPos);
  } else {
    return blocks.cend();
  }
}

virtual MemoryState compressAllMemory(const MemoryState &state) const final {
  auto currentState = state;

  while (true) {
    auto [blocks, freeBlocks] = currentState;
    // ищем первый свободный блок памяти
    // проверяем, есть ли за ним хотя бы один свободный блок
    uint32_t index = 0;
    for (; index < blocks.size() - 1 &&
           !(blocks[index].pid() == -1 && blocks[index + 1].pid() == -1);
         ++index) {
    }

    // если есть, то выполняем сжатие
    if (index < blocks.size() - 1) {
      currentState = compressMemory(currentState, index);
    } else {
      break;
    }
  }

  return currentState;
}

virtual MemoryState
allocateMemoryGeneral(const AllocateMemory &request,
                      const MemoryState &state,
                      const bool createProcess = false) const final {
  auto [blocks, freeBlocks] = state;

  // проверяем, выделены ли процессу какие-либо блоки памяти
  auto processPos = std::find_if(
      blocks.begin(), blocks.end(), [&request](const auto &block) {
        return block.pid() == request.pid();
      });
  // обработка некорректных ситуаций:
  // 1. Создание уже существующего процесса
  // 2. Выделение памяти несуществующему процессу
  if ((processPos != blocks.end() && createProcess) ||
      (processPos == blocks.end() && !createProcess)) {
    return state;
  }

  int32_t totalFree = 0;
  for (const auto &block : freeBlocks) {
    totalFree += block.size();
  }

  // проверяем, есть ли свободный блок подходящего размера
```

```
    // если есть, то выделяем процессу память в этом блоке
    if (auto pos = findFreeBlock(blocks, freeBlocks, request.pages());
        pos != blocks.end()) {
      uint32_t index = static_cast<uint32_t>(pos - blocks.cbegin());

      return allocateMemory(state, index, request.pid(), request.pages());
    } else if (totalFree >= request.pages()) {
      // если суммарно свободной памяти достаточно,
      // то выполняем дефрагментацию
      auto newState = defragmentMemory(state);
      auto [blocks, freeBlocks] = newState;
      auto pos = findFreeBlock(blocks, freeBlocks, request.pages());
      uint32_t index = static_cast<uint32_t>(pos - blocks.cbegin());

      return allocateMemory(newState, index, request.pid(), request.pages());
    } else {
      // недостаточно свободной памяти, игнорируем заявку
      return state;
    }
  }
};

using StrategyPtr = shared_ptr<AbstractStrategy>;

class FirstAppropriateStrategy final : public AbstractStrategy {
public:
  std::string toString() const override { return "FIRST_APPROPRIATE"; }

  static shared_ptr<FirstAppropriateStrategy> create() {
    return shared_ptr<FirstAppropriateStrategy>(new FirstAppropriateStrategy());
  }

protected:
  MemoryState sortFreeBlocks(const MemoryState &state) const override {
    auto currentState = state;
    auto [blocks, freeBlocks] = currentState;

    std::stable_sort(freeBlocks.begin(),
                     freeBlocks.end(),
                     [](const auto &left, const auto &right) {
                       return left.address() < right.address();
                     });

    return {blocks, freeBlocks};
  }

private:
  FirstAppropriateStrategy()
      : AbstractStrategy(StrategyType::FIRST_APPROPRIATE) {}
};

/**
 *  @brief Стратегия "Наиболее подходящий".
 */
class MostAppropriateStrategy final : public AbstractStrategy {
public:
  std::string toString() const override { return "MOST_APPROPRIATE"; }

  static shared_ptr<MostAppropriateStrategy> create() {
```

```
    return shared_ptr<MostAppropriateStrategy>(new MostAppropriateStrategy());
  }

private:
  MostAppropriateStrategy()
      : AbstractStrategy(StrategyType::MOST_APPROPRIATE) {}

protected:
  MemoryState sortFreeBlocks(const MemoryState &state) const override {
    auto currentState = state;
    auto [blocks, freeBlocks] = currentState;

    std::stable_sort(freeBlocks.begin(),
                     freeBlocks.end(),
                     [](const auto &left, const auto &right) {
                       if (left.size() == right.size()) {
                         return left.address() < right.address();
                       } else {
                         return left.size() < right.size();
                       }
                     });

    return {blocks, freeBlocks};
  }
};

class LeastAppropriateStrategy final : public AbstractStrategy {
public:
  std::string toString() const override { return "LEAST_APPROPRIATE"; }

  static shared_ptr<LeastAppropriateStrategy> create() {
    return shared_ptr<LeastAppropriateStrategy>(new LeastAppropriateStrategy());
  }

private:
  LeastAppropriateStrategy()
      : AbstractStrategy(StrategyType::LEAST_APPROPRIATE) {}

protected:
  MemoryState sortFreeBlocks(const MemoryState &state) const override {
    auto currentState = state;
    auto [blocks, freeBlocks] = currentState;

    std::stable_sort(freeBlocks.begin(),
                     freeBlocks.end(),
                     [](const auto &left, const auto &right) {
                       if (left.size() == right.size()) {
                         return left.address() < right.address();
                       } else {
                         return left.size() > right.size();
                       }
                     });

    return {blocks, freeBlocks};
  }
};
} // namespace MemoryManagement

#pragma once
```

```
#include <algorithm>
#include <cstdint>
#include <map>
#include <set>
#include <tuple>
#include <vector>

#include <nlohmann/json.hpp>

#include "exceptions.h"

namespace MemoryManagement {
class MemoryBlock {
private:
  int32_t _pid;

  int32_t _address;

  int32_t _size;

public:
  int32_t pid() const { return _pid; }

  int32_t address() const { return _address; }

  int32_t size() const { return _size; }

  MemoryBlock(int32_t pid, int32_t address, int32_t size)
      : _pid(pid), _address(address), _size(size) {
    validate(pid, address, size);
  }

  MemoryBlock(const MemoryBlock &other) = default;

  MemoryBlock() : MemoryBlock(-1, 0, 256) {}

  MemoryBlock &operator=(const MemoryBlock &rhs) = default;

  bool operator==(const MemoryBlock &rhs) const {
    return _pid == rhs._pid && _address == rhs._address && _size == rhs._size;
  }

  bool operator<(const MemoryBlock &rhs) const {
    return std::tuple{pid(), address(), size()} <
           std::tuple{rhs.pid(), rhs.address(), rhs.size()};
  }

  nlohmann::json dump() const {
    return {{"pid", _pid}, {"address", _address}, {"size", _size}};
  }

  static void validate(int32_t pid, int32_t address, int32_t size) {
    if (pid < -1 || pid > 255) {
      throw TypeException("INVALID_PID");
    }
    if (address < 0 || address > 255) {
      throw TypeException("INVALID_ADDRESS");
    }
```

```
    if (size < 1 || size > 256) {
      throw TypeException("INVALID_SIZE");
    }
    if (address + size > 256) {
      throw TypeException("OUT_OF_BOUNDS");
    }
  }
};

class MemoryState {
public:
  std::vector<MemoryBlock> blocks;

  std::vector<MemoryBlock> freeBlocks;

  /**
   *  @brief Создает дескриптор состояния памяти с заданными параметрами.
   *
   *  @param blocks Массив из дескрипторов всех доступных блоков памяти.
   *  @param freeBlocks Массив из дескрипторов свободных блоков памяти,
   *  упорядоченных согласно стратегии.
   */
  MemoryState(const std::vector<MemoryBlock> &blocks,
              const std::vector<MemoryBlock> &freeBlocks)
      : blocks(blocks), freeBlocks(freeBlocks) {}

  MemoryState() : MemoryState(MemoryState::initial()) {}

  MemoryState(const MemoryState &state) = default;

  MemoryState(MemoryState &&state) = default;

  MemoryState &operator=(const MemoryState &state) = default;

  MemoryState &operator=(MemoryState &&state) = default;

  bool operator==(const MemoryState &state) const {
    return blocks == state.blocks && freeBlocks == state.freeBlocks;
  }

  bool operator!=(const MemoryState &state) const { return !(*this == state); }

  nlohmann::json dump() const {
    auto jsonBlocks = nlohmann::json::array();
    auto jsonFreeBlocks = nlohmann::json::array();

    for (const auto &block : blocks) {
      jsonBlocks.push_back(block.dump());
    }
    for (const auto &block : freeBlocks) {
      jsonFreeBlocks.push_back(block.dump());
    }

    return {{"blocks", jsonBlocks}, {"free_blocks", jsonFreeBlocks}};
  }

  static MemoryState initial() {
    return {{MemoryBlock{-1, 0, 256}}, {MemoryBlock{-1, 0, 256}}};
  }
```

```
 static void validate(const std::vector<MemoryBlock> &blocks,
                      const std::vector<MemoryBlock> &freeBlocks) {
   if (blocks.size() == 0 && freeBlocks.size() == 0) {
     throw TypeException("INVALID_STATE");
   }

   // Проверяем, что множество свободных блоков freeBlocks и подмножество
   // свободных блоков в массиве blocks совпадают.
   std::set<MemoryBlock> set1(freeBlocks.begin(), freeBlocks.end()), set2;

   for (const auto &block : blocks) {
     if (block.pid() == -1) {
       set2.insert(block);
     }
   }

   if (set1 != set2) {
     throw TypeException("INVALID_STATE");
   }

   // Проверяем, что блоки памяти полностью покрывают адресное пространство.
   // Первый блок памяти должен начинаться с адреса 0.
   if (blocks.at(0).address() != 0) {
     throw TypeException("INVALID_STATE");
   }

   // Каждый блок, кроме последнего, должен заканчиваться там, где начинается
   // следующий.
   for (size_t i = 0; i < blocks.size() - 1; ++i) {
     auto &cur = blocks.at(i);
     auto &next = blocks.at(i + 1);
     if (cur.address() + cur.size() != next.address()) {
       throw TypeException("INVALID_STATE");
     }
   }

   // Последний блок должен полностью покрыть оставшееся пространство, при этом
   // не выходя за его границы.
   auto &last = blocks.back();
   if (last.address() + last.size() > 256) {
     throw TypeException("INVALID_STATE");
   }
 }
};
} // namespace MemoryManagement

#pragma once

#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <vector>

#include <tl/optional.hpp>

#include "types.h"

namespace ProcessesManagement {
```

```
inline tl::optional<std::size_t>
getIndexByPid(const std::vector<Process> &processes, int32_t pid) {

  auto pos =
      std::find_if(processes.begin(),
                   processes.end(),
                   [pid](const auto &process) { return process.pid() == pid; });

  if (pos == processes.end()) {
    return tl::nullopt;
  }

  return static_cast<std::size_t>(pos - processes.begin());
}

inline tl::optional<std::size_t> getIndexByPid(const ProcessesState &state,
                                               int32_t pid) {
  return getIndexByPid(state.processes, pid);
}

inline tl::optional<std::size_t>
getIndexByState(const std::vector<Process> &processes, ProcState state) {
  auto pos = std::find_if(
      processes.begin(), processes.end(), [state](const auto &process) {
        return process.state() == state;
      });

  if (pos == processes.end()) {
    return tl::nullopt;
  }

  return static_cast<std::size_t>(pos - processes.begin());
}

inline tl::optional<std::size_t> getIndexByState(const ProcessesState &state,
                                                 ProcState procState) {
  return getIndexByState(state.processes, procState);
}
} // namespace ProcessesManagement

#pragma once

#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <functional>
#include <map>
#include <set>
#include <vector>

#include "exceptions.h"
#include "helpers.h"
#include "types.h"

namespace ProcessesManagement {
inline ProcessesState changeProcessState(const ProcessesState &state,
                                         int32_t pid,
                                         ProcState newState) {
  auto [processes, queues] = state;
```

```
  if (auto index = getIndexByPid(processes, pid); index.has_value()) {
    processes.at(*index) = processes.at(*index).state(newState);
    return {processes, queues};
  } else {
    throw OperationException("NO_SUCH_PROCESS");
  }
}

inline ProcessesState
pushToQueue(const ProcessesState &state, size_t queueIndex, int32_t pid) {
  auto [processes, queues] = state;

  if (auto index = getIndexByPid(processes, pid); index.has_value()) {
    for (const auto &queue : queues) {
      if (std::find(queue.begin(), queue.end(), pid) != queue.end()) {
        throw OperationException("ALREADY_IN_QUEUE");
      }
    }
    queues.at(queueIndex).push_back(pid);
    processes.at(*index) = processes.at(*index).priority(queueIndex);
    return {processes, queues};
  } else {
    throw OperationException("NO_SUCH_PROCESS");
  }
}

inline ProcessesState popFromQueue(const ProcessesState &state,
                                   size_t queueIndex) {
  auto [processes, queues] = state;

  auto &queue = queues.at(queueIndex);
  if (queue.empty()) {
    throw OperationException("EMPTY_QUEUE");
  }

  auto pid = queue.front();

  if (auto index = getIndexByPid(processes, pid); !index.has_value()) {
    throw OperationException("NO_SUCH_PROCESS");
  }
  queue.pop_front();
  return {processes, queues};
}

inline ProcessesState switchTo(const ProcessesState &state, int32_t nextPid) {
  auto [processes, queues] = state;

  auto prevIndex = getIndexByState(processes, ProcState::EXECUTING);
  auto nextIndex = getIndexByPid(processes, nextPid);

  if (!nextIndex.has_value()) {
    throw OperationException("NO_SUCH_PROCESS");
  }

  auto next = processes.at(*nextIndex);
  if (prevIndex.has_value()) {
    auto prev = processes.at(*prevIndex);
    if (prev == next) {
```

```
      return state;
    } else {
      processes.at(*prevIndex) = prev.state(ProcState::ACTIVE);
    }

    if (next.state() != ProcState::ACTIVE) {
      throw OperationException("INVALID_STATE");
    }
  }
  processes.at(*nextIndex) = next.state(ProcState::EXECUTING);

  return {processes, queues};
}

inline ProcessesState terminateProcess(const ProcessesState &state,
                                       int32_t pid,
                                       bool terminateChildren = true) {
  auto [processes, queues] = state;

  if (auto index = getIndexByPid(processes, pid); !index.has_value()) {
    throw OperationException("NO_SUCH_PROCESS");
  }

  std::map<int32_t, std::vector<int32_t>> parents;
  for (const auto &process : processes) {
    if (process.ppid() != -1) {
      parents[process.ppid()].push_back(process.pid());
    }
  }
  std::set<int32_t> toTerminate;
  std::function<void(int32_t)> rec =
      [&rec, &toTerminate, &parents](int32_t pid) {
        toTerminate.insert(pid);
        for (const auto &child : parents[pid]) {
          rec(child);
        }
      };
  if (terminateChildren) {
    rec(pid);
  } else {
    toTerminate.insert(pid);
  }

  decltype(processes) newProcesses;
  decltype(queues) newQueues;

  for (const auto &process : processes) {
    if (toTerminate.find(process.pid()) == toTerminate.end()) {
      newProcesses.push_back(process);
    }
  }
  for (size_t i = 0; i < queues.size(); ++i) {
    const auto &queue = queues[i];
    auto &newQueue = newQueues[i];
    for (const auto &pid : queue) {
      if (toTerminate.find(pid) == toTerminate.end()) {
        newQueue.push_back(pid);
      }
    }
```

```
  }
  return {newProcesses, newQueues};
}

inline ProcessesState addProcess(const ProcessesState &state, Process process) {
  auto [processes, queues] = state;

  if (auto index = getIndexByPid(processes, process.pid()); index.has_value()) {
    throw OperationException("PROCESS_EXISTS");
  }

  auto parentIndex = getIndexByPid(processes, process.ppid());
  if (process.ppid() != -1 && !parentIndex.has_value()) {
    throw OperationException("NO_SUCH_PPID");
  }

  processes.push_back(process);
  std::sort(processes.begin(), processes.end());

  return {processes, queues};
}

inline ProcessesState updateTimer(const ProcessesState &state) {
  auto [processes, queues] = state;

  if (auto index = getIndexByState(processes, ProcState::EXECUTING);
      index.has_value()) {
    auto current = processes.at(*index);
    processes.at(*index) = current.timer(current.timer() + 1);
  }

  return {processes, queues};
}
} // namespace ProcessesManagement

#pragma once

#include <cstddef>
#include <cstdint>

#include <mapbox/variant.hpp>
#include <nlohmann/json.hpp>

#include "exceptions.h"
#include "types.h"

namespace ProcessesManagement {

class CreateProcessReq {
private:
  int32_t _pid;

  int32_t _ppid;

  size_t _priority;

  size_t _basePriority;

  int32_t _timer;
```

```
  int32_t _workTime;

public:
  CreateProcessReq(int32_t pid,
                   int32_t ppid = -1,
                   size_t priority = 0,
                   size_t basePriority = 0,
                   int32_t timer = 0,
                   int32_t workTime = 0)
      : _pid(pid), _ppid(ppid), _priority(priority),
        _basePriority(basePriority), _timer(timer), _workTime(workTime) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
    if (ppid < -1 || ppid > 255) {
      throw RequestException("INVALID_PPID");
    }
    if (priority > 15) {
      throw RequestException("INVALID_PRIORITY");
    }
    if (basePriority > 15 || basePriority > priority) {
      throw RequestException("INVALID_BASE_PRIORITY");
    }
    if (timer < 0) {
      throw RequestException("INVALID_TIMER");
    }
    if (workTime < 0) {
      throw RequestException("INVALID_WORK_TIME");
    }
  }

  int32_t ppid() const { return _ppid; }

  size_t priority() const { return _priority; }

  size_t basePriority() const { return _basePriority; }

  int32_t timer() const { return _timer; }

  int32_t workTime() const { return _workTime; }

  int32_t pid() const { return _pid; }

  nlohmann::json dump() const {
    return {{"type", "CREATE_PROCESS"},
            {"pid", _pid},
            {"ppid", _ppid},
            {"priority", _priority},
            {"basePriority", _basePriority},
            {"timer", _timer},
            {"workTime", _workTime}};
  }

  Process toProcess() const {
    return Process{}
        .pid(pid())
        .ppid(ppid())
        .priority(priority())
```

```
        .basePriority(basePriority())
        .timer(timer())
        .workTime(workTime());
  }
};

class TerminateProcessReq {
private:
  int32_t _pid;

public:
  TerminateProcessReq(int32_t pid) : _pid(pid) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
  }

  int32_t pid() const { return _pid; }

  nlohmann::json dump() const {
    return {{"type", "TERMINATE_PROCESS"}, {"pid", _pid}};
  }
};

class InitIO {
private:
  int32_t _pid;

public:
  InitIO(int32_t pid) : _pid(pid) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
  }

  int32_t pid() const { return _pid; }

  nlohmann::json dump() const { return {{"type", "INIT_IO"}, {"pid", _pid}}; }
};

class TerminateIO {
private:
  int32_t _pid;

  size_t _augment;

public:
  TerminateIO(int32_t pid, size_t augment = 1) : _pid(pid), _augment(augment) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }

    if (augment < 1 || augment > 15) {
      throw RequestException("INVALID_AUGMENT");
    }
  }

  int32_t pid() const { return _pid; }
```

```
  size_t augment() const { return _augment; }

  nlohmann::json dump() const {
    return {{"type", "TERMINATE_IO"}, {"pid", _pid}, {"augment", _augment}};
  }
};

class TransferControl {
private:
  int32_t _pid;

public:
  TransferControl(int32_t pid) : _pid(pid) {
    if (pid < 0 || pid > 255) {
      throw RequestException("INVALID_PID");
    }
  }

  int32_t pid() const { return _pid; }

  nlohmann::json dump() const {
    return {{"type", "TRANSFER_CONTROL"}, {"pid", _pid}};
  }
};

class TimeQuantumExpired {
public:
  TimeQuantumExpired() {}

  nlohmann::json dump() const { return {{"type", "TIME_QUANTUM_EXPIRED"}}; }
};

using Request = mapbox::util::variant<CreateProcessReq,
                                      TerminateProcessReq,
                                      InitIO,
                                      TerminateIO,
                                      TransferControl,
                                      TimeQuantumExpired>;
} // namespace ProcessesManagement

#pragma once

#include <array>
#include <cstddef>
#include <cstdint>
#include <deque>
#include <map>
#include <set>
#include <string>
#include <tuple>

#include <nlohmann/json.hpp>

#include "exceptions.h"

namespace ProcessesManagement {

enum class ProcState { ACTIVE, EXECUTING, WAITING };
```

```
class Process {
private:
  int32_t _pid;

  int32_t _ppid;

  size_t _priority;

  size_t _basePriority;

  int32_t _timer;

  int32_t _workTime;

  ProcState _state;

public:
  Process()
      : _pid(0), _ppid(-1), _priority(0), _basePriority(0), _timer(0),
        _workTime(0), _state(ProcState::ACTIVE) {}

  Process(const Process &other) = default;

  Process(Process &&other) = default;

  Process &operator=(const Process &rhs) = default;

  Process &operator=(Process &&rhs) = default;

  bool operator==(const Process &rhs) const {
    return std::tuple{_pid,
                      _ppid,
                      _priority,
                      _basePriority,
                      _timer,
                      _workTime,
                      _state} == std::tuple{rhs._pid,
                                            rhs._ppid,
                                            rhs._priority,
                                            rhs._basePriority,
                                            rhs._timer,
                                            rhs._workTime,
                                            rhs._state};
  }

  bool operator!=(const Process &rhs) const { return !(*this == rhs); }

  bool operator<(const Process &rhs) const {
    return std::tuple{_pid,
                      _ppid,
                      _priority,
                      _basePriority,
                      _timer,
                      _workTime,
                      _state} < std::tuple{rhs._pid,
                                           rhs._ppid,
                                           rhs._priority,
                                           rhs._basePriority,
                                           rhs._timer,
```

```
                                    rhs._workTime,
                                    rhs._state};
}

int32_t ppid() const { return _ppid; }

size_t priority() const { return _priority; }

size_t basePriority() const { return _basePriority; }

int32_t timer() const { return _timer; }

int32_t workTime() const { return _workTime; }

int32_t pid() const { return _pid; }

ProcState state() const { return _state; }

Process ppid(int32_t ppid) const {
  if (ppid < -1 || ppid > 255) {
    throw TypeException("INVALID_PPID");
  }

  Process other = *this;
  other._ppid = ppid;

  return other;
}

Process priority(size_t priority) const {
  if (priority > 15 || _basePriority > priority) {
    throw TypeException("INVALID_PRIORITY");
  }

  Process other = *this;
  other._priority = priority;

  return other;
}

Process basePriority(size_t basePriority) const {
  if (basePriority > 15 || basePriority > _priority) {
    throw TypeException("INVALID_BASE_PRIORITY");
  }

  Process other = *this;
  other._basePriority = basePriority;

  return other;
}

Process timer(int32_t timer) const {
  if (timer < 0) {
    throw TypeException("INVALID_TIMER");
  }

  Process other = *this;
  other._timer = timer;
```

```
    return other;
  }

  Process workTime(int32_t workTime) const {
    if (workTime < 0) {
      throw TypeException("INVALID_WORK_TIME");
    }

    Process other = *this;
    other._workTime = workTime;

    return other;
  }

  Process pid(int32_t pid) const {
    if (pid < 0 || pid > 255) {
      throw TypeException("INVALID_PID");
    }

    Process other = *this;
    other._pid = pid;

    return other;
  }

  Process state(ProcState state) const {
    Process other = *this;
    other._state = state;

    return other;
  }

  nlohmann::json dump() const {
    std::string state;
    switch (_state) {
    case ProcState::ACTIVE:
      state = "ACTIVE";
      break;
    case ProcState::EXECUTING:
      state = "EXECUTING";
      break;
    case ProcState::WAITING:
      state = "WAITING";
      break;
    }
    return {{"pid", _pid},
            {"ppid", _ppid},
            {"priority", _priority},
            {"basePriority", _basePriority},
            {"timer", _timer},
            {"workTime", _workTime},
            {"state", state}};
  }
};

class ProcessesState {
public:
  std::vector<Process> processes;
```

```
std::array<std::deque<int32_t>, 16> queues;

ProcessesState(const std::vector<Process> &processes,
               const std::array<std::deque<int32_t>, 16> &queues)
    : processes(processes), queues(queues) {}

ProcessesState(const std::vector<Process> &processes,
               const std::map<size_t, std::deque<int32_t>> &queues)
    : processes(processes), queues() {
  for (const auto &queue : queues) {
    this->queues.at(queue.first) = queue.second;
  }
}

ProcessesState() : ProcessesState(ProcessesState::initial()) {}

ProcessesState(const ProcessesState &state) = default;

ProcessesState(ProcessesState &&state) = default;

ProcessesState &operator=(const ProcessesState &state) = default;

ProcessesState &operator=(ProcessesState &&state) = default;

bool operator==(const ProcessesState &state) const {
  return processes == state.processes && queues == state.queues;
}

bool operator!=(const ProcessesState &state) const {
  return !(*this == state);
}

nlohmann::json dump() const {
  auto jsonProcesses = nlohmann::json::array();
  auto jsonQueues = nlohmann::json::array();

  for (const auto &process : processes) {
    jsonProcesses.push_back(process.dump());
  }
  for (const auto &queue : queues) {
    jsonQueues.push_back(queue);
  }

  return {{"processes", jsonProcesses}, {"queues", jsonQueues}};
}

static ProcessesState initial() {
  return {{}, std::array<std::deque<int32_t>, 16>{}};
}

static void validate(const std::vector<Process> &processes,
                     const std::array<std::deque<int32_t>, 16> &queues) {
  std::set<int32_t> pidsInQueues, pidsOfProcesses, activePids;
  std::map<size_t, std::set<int32_t>> queuesMap;

  // В списке процессов не должно быть двух процессов с одинаковым PID.
  for (const auto &process : processes) {
    if (pidsOfProcesses.find(process.pid()) != pidsOfProcesses.end()) {
      throw TypeException("INVALID_STATE");
```

```
      } else {
        pidsOfProcesses.insert(process.pid());
      }
    }

    // Собираем список процессов в состоянии ACTIVE.
    for (const auto &process : processes) {
      if (process.state() == ProcState::ACTIVE) {
        activePids.insert(process.pid());
      }
    }

    // Проверяем на существование родительских процессов.
    for (const auto &process : processes) {
      if (process.ppid() != -1 &&
          pidsOfProcesses.find(process.ppid()) == pidsOfProcesses.end()) {
        throw TypeException("INVALID_STATE");
      }
    }

    // Собираем множество процессов, находящихся в очередях.
    for (size_t i = 0; i < queues.size(); ++i) {
      const auto &queue = queues[i];
      for (const auto &pid : queue) {
        // Каждый процесс должен находится только в одной очереди.
        if (pidsInQueues.find(pid) != pidsInQueues.end()) {
          throw TypeException("INVALID_STATE");
        } else {
          pidsInQueues.insert(pid);
        }
        queuesMap[i].insert(pid);
      }
    }

    // Только процессы с состоянием ACTIVE могут находится в очередях.
    if (activePids != pidsInQueues) {
      throw TypeException("INVALID_STATE");
    }

    // Проверяем процессы на соответствие приоритета и индекса очереди.
    for (const auto &process : processes) {
      if (process.state() != ProcState::ACTIVE) {
        continue;
      }
      auto priority = process.priority();
      const auto &queueSet = queuesMap[priority];
      if (queueSet.find(process.pid()) == queueSet.end()) {
        throw TypeException("INVALID_STATE");
      }
    }
  }
};
} // namespace ProcessesManagement
```