Universidade Federal do Ceará - Campus Quixadá
QXD0010 – Estruturas de Dados
Turma 02A – Ciência da Computação
Prof. Atílio Gomes Luiz

Erica de Castro e Kauan Pablo

06/11/2023

Relatório do Sistema Neolook

Repositório do projeto no GitHub

Equipe

- Erica de Castro Silveira, 552460 - GitHub
- Kauan Pablo de Sousa Silva, 556027 -GitHub

01. Introdução

O projeto consiste em uma simulação que tem como objetivo avaliar o desempenho de duas políticas de escalonamento, FCFS (First-Come-First-Served) e SJF (Shortest Job First), em um sistema distribuído chamado NeoLook. Este sistema é composto por N computadores interligados através de uma rede de alta velocidade, sendo que cada computador está equipado com uma CPU e dois discos. Quando um processo é submetido para execução, ele percorre uma série de etapas, incluindo a fase de processamento na CPU, o acesso a um dos discos e a transferência de dados pela rede.

O principal objetivo da simulação é reproduzir a entrada de um número específico de processos no sistema e, a partir dessa simulação, calcular métricas como o tempo médio de execução dos processos, o tempo médio de espera dos processos e a taxa de processamento do sistema, métricas que serão utilizadas para avaliar o desempenho do sistema nas duas políticas de escalonamento.

02. Estruturas de Dados utilizadas

Foram implementadas duas estruturas de dados para representar as filas de espera, uma Queue e uma Min Heap, essas estruturas desempenham um papel fundamental no gerenciamento dos processos e na ordem de execução, garantindo que os processos sejam tratados de acordo com as políticas de escalonamento FCFS (First Come, First Served) ou SJF (Shortest Job First), proporcionando uma organização eficaz e justa na execução de tarefas, seja seguindo a ordem de chegada ou priorizando as tarefas mais curtas para otimizar o desempenho do sistema.

02.1 Fila (Queue)

Uma fila (queue) é uma estrutura de dados que segue o princípio FIFO (First In, First Out), onde o primeiro elemento inserido é o primeiro a ser removido. Ela é composta por nós que armazenam valores e possuem ponteiros para os nós seguintes, a classe queue implementa essa estrutura. A seguir estão as três principais operações da Queue:

- `push(const type &elem)`: O método `push` insere um elemento no final da fila fazendo as seguintes operações:

```
// Insere um elemento no final da fila.
// Complexidade O(1) (Operação Constante)
void push(const type &elem) {
    if (m_first == nullptr) {
        // Se a fila estiver vazia, criamos um novo nó e fazemos o
        //  ponteiro para o primeiro e último nó apontar para ele.
        m_last = m_first = new node<type>(elem, nullptr);
    } else {
        // Se a fila não estiver vazia, criamos um novo nó e fazemos
        // o ponteiro para o último nó apontar para ele.
        m_last->get_next() = new node<type>(elem, nullptr);
        m_last = m_last->get_next();
    }
    // Atualizamos o tamanho da fila.
    m_size++;
}
```

- `pop()`:

  O método `pop` remove o elemento do início da fila, também seguindo o princípio FIFO (First In, First Out). Se a fila estiver vazia, ele retorna nulltpr. Caso contrário, a função realiza as seguintes etapas para efetuar a remoção do elemento e retorna o elemento que foi removido:

```
// Remove o elemento do início da fila.
// Complexidade O(1) (Operação constante)
type pop() {
if (m_size == 0) {
    // Se a fila estiver vazia, retornamos nullptr.
    return nullptr;
}

// Criamos um nó auxiliar para não perder o ponteiro para o
// primeiro nó,e fazemos o ponteiro para o primeiro nó apontar
// para o segundo nó.
node<type> *aux = m_first;
type key = aux->get_value();
m_first = m_first->get_next();

// Fazemos o ponteiro para o próximo nó do nó auxiliar apontar
// para nullptr e desalocamos a memória do nó auxiliar.
aux->get_next() = nullptr;
delete aux;

// Atualizamos o tamanho da fila, e se a fila ficou vazia,
// atualizamos o ponteiro para o último nó.
m_size--;
if (m_size == 0) {
    m_last = nullptr;
}

// Retornamos o valor do nó removido.
return key;
}
```

- `front()`:

  O método `front` retorna o elemento do início da fila. Se a fila estiver vazia, o método lança uma exceção. Esta função é útil para verificar o valor do elemento na frente da fila antes de removê-lo.

```
// Retorna o elemento do início da fila. O(1)
type &front() {
    if (m_size == 0) {
        throw std::runtime_error("fila vazia");
    }
    return m_first->get_value();
}
```

02.2 Fila de Prioridade (Min Heap)

Uma fila de prioridade é uma estrutura de dados que permite armazenar elementos associados a prioridades e garantir que os elementos com maior prioridade sejam processados primeiro. Nesse projeto, a fila de prioridade é implementada como uma min heap. Utilizada para simular a política de escalonamento de processos Shortest Job First (SJF), onde o processo com menor tempo de execução é executado primeiro.

A min heap é um tipo de heap binário baseado em árvore, que funciona da seguinte forma: O valor de cada elemento é menor ou igual ao valor de seus filhos. A raiz da árvore é o elemento de menor valor. A min heap é uma estrutura de dados eficiente para implementar uma fila de prioridade porque permite inserir e remover elementos em tempo O(log n) na maioria dos casos.

Aqui estão as principais funções da Min Heap:

- `push(type value)`: Insere um elemento na fila de acordo com sua prioridade. A operação de inserção pode envolver a reorganização da árvore (subida na heap) para manter a propriedade da min heap, para isso, utilizamos o método `up`.

```
// Insere um elemento na heap.
// Complexidade O(n) no pior caso (capacidade cheia) pois
// será necessário a chamada da função reserve e log(n)
// no resto dos casos, pois a função up é log(n).
void push(type value) {
    // Insere elemento no final do heap e incrementa o tamanho.
    ptr[m_size] = value;
    m_size++;

    // Dobra a capacidade do vetor se necessário. O(n)
    if (m_size == m_capacity) {
        reserve();
    }

    // Chama a função up para colocar o elemento na posição correta.
    // O(log n)
    up(m_size - 1);
}
```

- `pop()`: Remove o elemento de maior prioridade da fila (menor elemento), que é o elemento na raiz da min heap. Após a remoção, a árvore é reorganizada (descida na heap) utilizando o método `down` para manter a propriedade da min heap e no final é retornado o elemento que foi removido.

```
// Remove o elemento da raiz da heap.
// Complexidade log(n) em todos os casos pois a função down é log(n).
type pop() {
    // Verifica se a heap está vazia, e retorna nullptr se estiver.
    if (m_size == 0) {
        return nullptr;
    }

    // Troca o elemento da raiz com o último elemento da heap e
    // decrementa o tamanho.
    swap(ptr[0], ptr[m_size - 1]);
    m_size--;

    // Chama a função down para colocar o novo elemento da raiz na
    // posição correta. O(log n)
    down(0);

    return ptr[m_size];
}
```

Antes de analisar os métodos `up` e `down` um método que vale resaltar é o `set_comparator()`.

```
void set_comparator(int (*comparator)(type, type)) {
    this->comparator = comparator;
}
```

Como a fila de prioridade representa um recurso, seja fila de espera da cpu, disco ou rede e nela serão inseridos processos, logo é necessário setar qual atributo de um processo será comparado, o que será mais explicado na seção de processos.

Agora que isso foi mencionado, além das operações principais, a min heap requer dois métodos essenciais para manter sua propriedade:

- `up(int index)`: O método up move um elemento para cima na heap, garantindo que o pai seja menor que os filhos. Isso é necessário após a inserção de um elemento. A função é chamada com o índice do elemento recém-inserido e, se necessário, troca o elemento com seu pai e chama recursivamente up para o pai.

```
    // Move um elemento para cima na heap, garantindo que o pai seja menor
    // que os filhos.

    // Complexidade O(log n) em todos os casos pois o elemento percorre a
    // altura da árvore binária, e a altura de uma heap é limitada pelo
    // logaritmo do número de elementos (n)
    void up(int index) {
        // Verifica se o elemento é a raiz, pois se for, não tem pai.
        if (index == 0) {
            return;
        }

        // Calcula o índice do pai.
        int daddy = (index - 1) / 2;

        // Verifica se o pai é maior que seu filho.
        // Se for, troca o pai com o filho e chama a função up para o pai,
        // pois o pai pode ser maior que o seu pai.
```

```
        if (comparator(ptr[daddy], ptr[index]) == 1) {
            swap(ptr[daddy], ptr[index]);
            up(daddy);
        }
    }
```

- `down(int index)`: O método `down` move um elemento para baixo na heap, garantindo que o filho seja maior que o pai. Isso é necessário após a remoção do elemento de maior prioridade. A função é chamada com o índice do elemento removido, e se necessário, troca o elemento com seu filho de menor valor e chama recursivamente `down` para o filho.

```
// Move um elemento para baixo na heap, garantindo que o filho seja
// maior que os pais. O(log n)
void down(int index) {
    // Calcula o índice do filho esquerdo, e assume que o filho
    // esquerdo é o menor filho.
    int index_smaller = (index * 2) + 1;

    // Verifica se existe filho esquerdo.
    if (index_smaller >= this->m_size) {
        return;
    }

    // Verifica se existe o filho direito.
    // Se existir, verifica se o filho direito é menor que o
    // filho esquerdo e atualiza o índice do menor filho.
    if ((index_smaller + 1) < this->m_size) {
        if (comparator(ptr[index_smaller],
            ptr[index_smaller + 1]) == 1) {
            index_smaller += 1;
        }
    }

    // Verifica se o elemento i (pai) é maior que o menor filho.
    // Se for, troca o elemento i com o menor filho e chama a
    // função down para o menor filho, pois o menor filho pode
    // ser maior que os seus filhos.
    if (comparator(ptr[index], ptr[index_smaller]) == 1) {
        swap(ptr[index], ptr[index_smaller]);
        down(index_smaller);
    }
}
```

02.3 Conclusão e observação das estruturas de dados

Entender o básico das estruturas de dados é importante para entender algumas partes da lógica das filas de espera dentro do escalonador, conforme as classes do sistema forem sendo explicadas partes mais específicas das estruturas serão abordadas. Lembrando que as estruturas foram otimizadas ao máximo para evitar métodos que não serão usados no sistema, e para que as operações com ambas as filas possam ocorrer corretamente independente do tipo de fila os métodos principais tem os mesmos nomes, como no caso do `push()`, `pop()`, `empty()`.

03. Divisão das tarefas

- Erica desempenhou um papel fundamental na implementação das estruturas de dados. Ela, em particular, foi a responsável pela implementação das duas estruturas e em destaque a Min Heap.
- Kauan focou em revisar e refatorar o código, garantindo a qualidade e a organização do projeto. Ele também assumiu a responsabilidade de cuidar da estrutura geral do projeto, assegurando que os componentes se integrassem de forma correta.
- Trabalho Conjunto no Sistema Neolook: O desenvolvimento do Sistema Neolook foi um esforço conjunto, com ambas as partes contribuindo para sua funcionalidade. Embora as tarefas não tenham sido rigidamente divididas, a maior parte da lógica do sistema e implementações foram feitas pelo trabalho da Erica, enquanto a organização e implementação de partes mais específicas do sistema se deu pelo Kauan.
- Relatórios: A responsabilidade pela elaboração dos relatórios recaiu sobre Kauan, que cuidou da documentação do projeto.

04. Implementação do Sistema

Para compreender o funcionamento do sistema é importante conhecermos todas as classes do sistema e como elas se relacionam, as classes serão explicadas conformem forem se relacionando, primeiro uma explicação básica da classe, um esboço da mesma e se preciso a explicação de algum método relevante junto com sua implementação. As classes implementadas foram as seguintes:

04.1 Processo (process.h)

Um processo é basicamente um objeto que vai guardar 5 valores, o seu ID, instante que ele chegou no sistema, e as demandas que ele tem de cpu, disco e rede, a classe foi implementada da seguinte forma:

```
#ifndef PROCESS_H
#define PROCESS_H

// Classe que armazena um processo.
class process {
   public:
    int id;              // Identificador do processo.
    int instant;         // Instante de chegada do processo.
    int demand_cpu;      // Demanda de CPU do processo.
    int demand_disk;     // Demanda de disco do processo.
    int demand_network;  // Demanda de rede do processo.

    // Construtor, inicializa os atributos.
    process(int id, int instant, int demand_cpu, int demand_disk,
            int demand_network) {
        // ...
    }

    // Printa o processo.
    void print() {
       // ...
    }
};

/**
```

```
 * Uma breve explicação sobre os comparadores:
 *
 * Os comparadores são funções que comparam dois processos e retornam um inteiro
 * que indica a relação entre os dois processos, eles são utilizados para
 * ordenar os processos nas filas de prioridade, já que a min_heap precisa de
 * algum parâmetro para ordenar os processos que depende de onde o processo
 * está.
 *
 */

// Comparador de processos em relação a demanda de CPU.
int compare_process_cpu(process* a, process* b) {
    if (a->demand_cpu < b->demand_cpu) {
        return -1;
    } else if (a->demand_cpu > b->demand_cpu) {
        return 1;
    } else {
        return 0;
    }
}

// Comparador de processos em relação a demanda de disco.
int compare_process_disk(process* a, process* b) {
    if (a->demand_disk < b->demand_disk) {
        return -1;
    } else if (a->demand_disk > b->demand_disk) {
        return 1;
    } else {
        return 0;
    }
}

// Comparador de processos em relação a demanda de rede.
int compare_process_network(process* a, process* b) {
    if (a->demand_network < b->demand_network) {
        return -1;
    } else if (a->demand_network > b->demand_network) {
        return 1;
    } else {
        return 0;
    }
}

#endif
```

Além desses atributos ele tem três métodos que servem para definir qual parâmetro a min_heap vai utilizar para ordenar os processos nas respectivas filas de espera (CPU, disco e rede).

04.2 Computador (computer.h)

A classe computer representa um computador dentro do sistema, ele tem três filas de espera(o tipo de fila é definida no construtor, min heap ou queue) e os ponteiros para os processos que estão em execução em algum lugar do computador, ela está implementada da seguinte forma:

```
#ifndef COMPUTER_H
#define COMPUTER_H
```

```
#include <iostream>

#include "../include/min_heap.h"
#include "../include/queue.h"
#include "process.h"

// Classe que armazena um computador.
using namespace std;
template <typename Type>
class computer {
   public:
    Type* cpu;                // Fila de espera da CPU.
    Type* disk_1;             // Fila de espera do disco 1.
    Type* disk_2;             // Fila de espera do disco 2.
    process* running_cpu;     // Processo em execução na CPU.
    process* running_disk_1;  // Processo em execução no disco 1.
    process* running_disk_2;  // Processo em execução no disco 2.

    // Construtor, inicializa as filas e os processos em execução.
    computer() {
        // ...
    }

    // Destrutor, desaloca as filas e os processos em execução.
    ~computer() {
        // ...
    }

    // Verifica se existe algum processo em execução ou nas filas.
    bool has_process() {
       // ...
    }

    // Verifica se existe algum processo em execução na CPU.
    bool has_process_in_cpu() {
        // ...
    }

    // Verifica se existe algum processo em execução no disco 1.
    bool has_process_in_disk_1() {
       //
    }

    // Verifica se existe algum processo em execução no disco 2.
    bool has_process_in_disk_2() {
        // ..
    }

    // Verifica se existe algum processo em execução na rede.
    void add_process_cpu(process* process) {
        // ...
    }

    // Adiciona um processo na fila de espera de um dos discos.
    void add_process_disk() {
```

```
        // ...
    }

};

#endif
```

A classe contém uma série de métodos para verificar a existência de processos em locais específicos do computador, bem como métodos para adicionar processos à fila de espera, seja na CPU ou na rede, além do método para consumir uma unidade de tempo do processo na CPU. Dentre esses, dois métodos merecem destaque:

- `add_process_cpu()`: O método recebe um ponteiro para um processo e coloca esse processo na fila de espera da CPU.

  ```
  // Adiciona um processo na fila de espera da CPU.
  void add_process_cpu(process* process) {
      cpu->push(process);
  }
  ```

- `add_process_disk()` : O método move o processo que está em execução na cpu(caso exista) e adiciona ele na fila de espera de um disco aleatório do computador.

  ```
  // Adiciona um processo na fila de espera de um dos discos.
  void add_process_disk() {
      // Verifica se existe algum processo em execução na CPU.
      if (this->running_cpu != nullptr) {
          // Cria um número aleatório entre 0 e 1.
          int random = rand() % 2;

          // Com base no número aleatório, adiciona o processo na
          // fila de espera do disco 1 ou 2.
          if (random == 0) {
              disk_1->push(this->running_cpu);
          } else {
              disk_2->push(this->running_cpu);
          }
      }

      // Limpa o processo em execução na CPU.
      this->running_cpu = nullptr;
  }
  ```

04.3 Recurso (resource.h)

Um recurso é composto de N computadores, uma fila de espera para a rede e o ponteiro para o processo que está sendo executado na rede. A classe resource foi implementada da seguinte forma:

```
#ifndef RESOURCE_H
#define RESOURCE_H

#include "../include/min_heap.h"
#include "../include/queue.h"
#include "computer.h"
```

```
#include "enumerator.h"
#include "process.h"

template <typename type>
class resource {
   public:
    computer<type> *computer_type;  // Vetor de computadores.
    int amount;                     // Quantidade de computadores.
    priority_policy policy;         // Política de escalonamento. (Enumerator)
    type *network;                  // Fila de espera da rede.
    process *running_network;       // Processo que está sendo executado na rede.

    // Construtor, inicializa os computadores e a rede.
    resource(int amount, priority_policy policy) {
        // ...
    }

    // Adiciona um processo em uma das CPUs.
    void add_process(process *process) {
        // ..
    }

    // Verifica se existe processo na rede.
    bool has_process_in_network() {
        // ...
    }

    // Destrutor
    ~resource() {
        // ...
    }
};

#endif
```

O método que merece mais destaque dessa classe é o `add_process()`, ele recebe um ponteiro para um processo e o coloca para ser executado em algum computador aleatório do sistema.

```
// Adiciona um processo em uma das CPUs.
void add_process(process *process) {
    // Cria um index aleatório.
    int random_index = rand() % amount;
    // Adiciona o processo na CPU do index aleatório.
    computer_type[random_index].add_process_cpu(process);
}
```

Além desse método devemos dar analisar o construtor da classe:

```
// Construtor, inicializa os computadores e a rede.
resource(int amount, priority_policy policy) {
    this->amount = amount;
    this->policy = policy;
    this->computer_type = new computer<type>[amount];
    this->network = new type();
    this->running_network = nullptr;
```

```
    // Caso a política seja SJF, seta os comparadores da min_heap nas respectivas
    // filas.
    if (policy == SJF) {
        // Para cada computador, seta o comparador da CPU e dos discos.
        for (int i = 0; i < amount; i++) {
                this->computer_type[i].cpu->set_comparator(
                    &compare_process_cpu);
                this->computer_type[i].disk_1->set_comparator(
                    &compare_process_disk);
                this->computer_type[i].disk_2->set_comparator(
                    &compare_process_disk);
            }
        }

        // Setando o comparador da rede.
        this->network->set_comparator(&compare_process_network);
    }
}
```

Além de inicializar normalmente seus atributos, caso a política escolhida seja SJF (ou seja, vai ocorrer a utilização da min heap) ele precisa percorrer o vetor de computadores e chama a função `set_comparator()` da min heap usando como argumento qual comparador ela vai utilizar(visto na classe process.h) além de no final setar o comparador de rede.

04.4 Eventos (event.h)

Antes da explicação do funcionamento do escalonador é necessário o entendimento dos eventos do sistema. Um evento é uma classe que guarda o instante em que ele ocorreu, o tipo de evento e o processo associado a ele. A classe foi implementada da seguinte forma:

```
#ifndef EVENT_H
#define EVENT_H
#include "enumerator.h"
#include "process.h"

// Classe que armazena os eventos do simulador.
class event {
   public:
    int instant;        // Instante em que o evento ocorre.
    type_event type;    // Tipo do evento.
    process* process_a;  // Processo associado ao evento.

    // Construtor, inicializa os atributos.
    event(int instant, type_event event, process* p) {
        // ...
    }

    // Destrutor, deleta o processo.
    ~event() {
        // ...
    }

    // Printa o evento. (Utilizado para debug)
    void print() {
        // ...
    }
```

```
};

#endif
```

Os tipos de eventos estão salvos no seguinte enumerador:

```
// Enumerador que armazena os tipos de eventos.
typedef enum type_event {
    INITIALIZE_PROCESS,
    WAIT_CPU,
    ACCESS_CPU,
    WAIT_DISK,
    ACCESS_DISK,
    WAIT_NETWORK,
    ACCESS_NETWORK,
    FINISH_PROCESS
} type_event;
```

O registro de eventos que ocorrem no sistema serão de extrema importância para calcular as métricas do escalonador, o que será visto um pouco mais a frente.

04.5 Metricas (metrics.h)

Classe usada para criar um objeto que vai guardar os cálculos feitos pelo escalonador, printar e salvar em um arquivo as estatíscas. Ela está implementada da seguinte forma:

```
#ifndef METRICS_H
#define METRICS_H

#include <iomanip>
#include <iostream>
#include <fstream>
#include <string>

#include "event.h"

// Classe que armazena as métricas de execução do simulador.
class metrics {
   private:
    int total_time;          // Tempo total de execução.
    double average_time;     // Tempo médio de execução.
    double average_wait;     // Tempo médio de espera.
    double processing_rate;  // Taxa de processamento.

   public:
    // Construtor, calcula e inicializa as métricas.
    metrics(int total_time, int num_process, event*** log) { {
         // ...
    }

    // Printa as métricas.
    void print() {
        // ...
    }

    // Salvando as métricas em um arquivo.
```

```
    void save(string filename, int num_computers, string policy) {
    // ...

};

#endif
```

Para entender o cálculo das métricas, é importante olhar o funcionamento do construtor, além de recer o tempo total de execução do sistema e o número de processos, ele receve uma matriz de ponteiros para eventos, Cada evento está em sua devida linha de processo de acordo com o ID do processo e os eventos de cada processo estão em ordem cronológica.(Matriz feita na classe escalonator que será a proxima a classe a ser explicada.)

```
// Construtor, calcula e inicializa as métricas.
metrics(int total_time, int num_process, event*** log) {
// Tempo medio de resposta = (tempo de espera + tempo dexecucao) /
//                              numero de processos
double average_time = 0;

// Tempo medio de espera = tempo de espera / numero dprocessos
double average_wait = 0;

// Taxa de processamento = numero de processos /
//     termino dultimo processo - inicio do primeiro processo
double processing_rate = static_cast<double>(num_process) /
                            (total_time - log[0][0]->instant);

// Percorrendo cada processo.
for (int i = 0; i < num_process; i++) {
    // Demanda da CPU de quando o processo foi inicializado.
    double demand_cpu = log[i][0]->process_a->demand_cpu;

    // Demanda do disco de quando o processo foi inicializado.
    double demand_disk = log[i][0]->process_a->demand_disk;

    // Demanda da rede de quando processo foi inicializado.
    double demand_network = log[i][0]->process_a->demand_network;

    // Tempo de espera na CPU.
    // (instante de acesso a CPU - instante de espera da CPU)
    double wait_cpu = log[i][2]->instant - log[i][1]->instant;

    // Tempo de espera no disco.
    // (instante de acesso ao disco - instante de espera do disco)
    double wait_disk = log[i][4]->instant - log[i][3]->instant;

    // Tempo de espera na rede.
    // (instante de acesso a rede - instante de espera da rede)
    double wait_network = log[i][6]->instant - log[i][5]->instant;

    // Tempo de execucao = tempo de espera geral + demanda geral
    average_time += (wait_cpu + wait_disk + wait_network +
                        demand_cpu + demand_disk + demand_network);

    // Tempo de espera = tempo de espera na CPU + tempo de espera no disco +
    //                   tempo de espera na rede
```

```
    average_wait += (wait_cpu + wait_disk + wait_network);
}
// Tempo medio de execucao = tempo medio de execucao / numero de processos
average_time /= num_process;
// Tempo medio de espera = tempo medio de espera / numero de processos
        average_wait /= num_process;

// Setando as métricas.
this->total_time = total_time;
this->average_time = average_time;
this->average_wait = average_wait;
this->processing_rate = processing_rate;}
```

04.6 Escalonador (escalonator.h)

A classe que vai simular a execução dos processos dentro do recurso e salvar os eventos que ocorrem.

```
#ifndef ESCALONATOR_H
#define ESCALONATOR_H

#include "../include/queue.h"
#include "computer.h"
#include "enumerator.h"
#include "event.h"
#include "metrics.h"
#include "resource.h"

// Classe que representa o escalonador.
template <typename Type>
class escalonator {
   public:
    queue<event*> events;            // Fila de eventos
    resource<Type>* system;          // Sistema(um recurso)
    queue<process*>* process_queue;  // Fila de processos

    // Vetor de ponteiros para eventos (serao registrados 8 eventos por
    // processo)
    event** events_vec;
    int size_events;  // Tamanho do vetor de eventos
    int qtd_process;  // Quantidade de processos

    // Matriz de ponteiros para eventos, cada linha eh dedicada a um processo
    // possuindo todos os eventos relacionados a ele.
    event*** log;

    // Construtor (recebe a quantidade de computadores, a politica de
    // escalonamento e a fila de processos a serem processados).
    escalonator(int amount, priority_policy p, queue<process*>* process_queue) {
        // ...
    }

    // Metodo que executa o escalonamento.
    void run(string filename) {
        // ...
```

```
    }

    // Destrutor, desaloca os eventos, o sistema e a matriz de eventos.
    ~escalonator() {
        // ...
    }
};

#endif
```

Os atributos dessa classe incluem:

- Uma fila de eventos que é usada para salvar a ordem de registro dos eventos.
- Um recurso que representa o sistema.
- Uma fila de processos contendo os processos que serão inicializados no sistema.
- Um vetor de ponteiros para eventos (serão registrados 8 eventos por processo)
- Uma matriz de ponteiros para eventos, onde cada linha é dedicada a um processo e contém todos os eventos relacionados a ele.
- A quantidade de processos e o tamanho do vetor de eventos.

O construtor da classe está implementado da seguinte forma:

```
// Construtor (recebe a quantidade de computadores, a politica de
// escalonamento e a fila de processos a serem processados).
escalonator(int amount, priority_policy p, queue<process*>* process_queue) {
    this->system = new resource<Type>(amount, p); // Inicializando o sistema
    this->process_queue = process_queue; // Inicializando a fila de processos
    // Inicializando a quantidade de processos
    qtd_process = process_queue->size();

    // Cada processo sera registrado em 8 eventos:
    //      -> Iniciar processo.
    //      -> Esperar CPU
    //      -> Acessar CPU
    //      -> Esperar Disco
    //      -> Acessar Disco
    //      -> Esperar Rede
    //      -> Acessar Rede
    //      -> Terminar Processo

    // O vetor de eventos tera o tamanho de 8 vezes a quantidade de
    // processos.
     events_vec = new event*[qtd_process * 8];

    // Inicializando o tamanho do vetor de eventos.
    size_events = 0;

    // Inicializando a matriz de eventos.
    log = new event**[qtd_process];

    // Inicializando cada linha da matriz de eventos.
    for (int i = 0; i < qtd_process; i++) {
        log[i] = new event*[8];
```

```
    }
}
```

O método que executa o escalonamento é o método `run()`, que será divido em partes para facilitar a explicação.

- **1. Execução do escalonador.**

  O primeiro loop é onde ocorre o escalonamento, ele contém um contador para simular um relógio lógico, e é aqui onde todos os processos serão colocados para a execução, processados, e os demais eventos que um processo pode passar.

```
int time = 0;  // Instante de tempo

// Loop que executa o escalonamento.
while (true) {
    //...
    time++;
}
```

- **1.1 Condição de parada do escalonador.**
  O trecho inicial do código, localizado dentro do loop, realiza uma série de verificações com o objetivo de determinar se ainda existem processos no sistema ou se há a possibilidade de adição de processos no futuro. Caso nenhuma dessas condições seja satisfeita, o escalonador encerra a execução do sistema.

```
// Flag que indica se ainda ha processos no sistema.
bool has_process_in_system = false;

// Verificando se ha processo a ser processado no sistema.
for (int i = 0; i < this->system->amount; i++) {
    // Se houver processo no sistema, a flag eh setada para true e o
    // loop eh quebrado.
    if (this->system->computer_type[i].has_process()) {
        has_process_in_system = true;
        break;
    }
}

// Se nao houver processo na fila de processos, no sistema e na
// rede, o escalonamento eh finalizado.
if (process_queue->empty() &&
    !this->system->has_process_in_network() &&
    !has_process_in_system) {
    break;
}
```

- **1.2 Adição de processos no sistema.**
  O segundo bloco de código transfere os processos da fila de processos (caso ela não esteja vazia) e os incorpora ao sistema, verificando se o instante de inicialização coincide com o tempo atual do sistema (indicado pelo contador de tempo). A inicialização do processo é registrada nos eventos, e o evento de espera da CPU é agendado simultaneamente à inicialização do processo no sistema.

```
// Enquanto houver processo na fila de processos e o instante de
// tempo for igual ao instante de chegada do processo, o processo eh
// adicionado ao sistema.
while (!process_queue->empty() && process_queue->front()->instant == time) {
    // Retirando o processo da fila de processos
    process* p = process_queue->pop();

    // Registrando o evento de inicializacao do processo.
    events_vec[size_events] = (new event
        (time, INITIALIZE_PROCESS, new process
        (p->id, p->instant, p->demand_cpu,
        p->demand_disk, p->demand_network)));

    // Incrementando o tamanho do vetor de eventos.
    size_events++;

    // Registrando o evento de espera da CPU.
    events_vec[size_events] = (new event
        (time, WAIT_CPU, new process
        (p->id, p->instant, p->demand_cpu,
        p->demand_disk, p->demand_network)));

     // Incrementando o tamanho do vetor de eventos.
     size_events++;

    // Adicionando o processo ao sistema.
    system->add_process(p);
}
```

- **1.3 Executando os processos.**
  O bloco de código a seguir contém um loop que itera por todos os computadores, gerenciando a execução de processos e suas transferências entre recursos. Ele registra eventos sempre que um recurso começa a ser executado na CPU, entra na fila de espera do disco, inicia a execução pelo disco, entra na fila de espera da rede, inicia a execução na rede e quando o processo é finalizado.

```
// Percorrendo cada computador do sistema.
for (int i = 0; i < system->amount; i++) {
    // Criando um ponteiro para o computador atual.
    computer<Type>* pc = &this->system->computer_type[i];

    // PROCESS_CPU
    // Verificando se nao existe processo em execucao na CPU.
        if (!pc->has_process_in_cpu()) {
            // Retirando o processo da fila de espera da CPU.
            process* p = pc->cpu->pop();

            // Verificando se existe processo na fila de espera da CPU.
            if (p != nullptr) {
                // Se existir, o processo eh adicionado a execucao da CPU.
                pc->running_cpu = p;

                // Registrando o evento de acesso a CPU.
                events_vec[size_events] = (new event
```

```
            (time, ACCESS_CPU, new process
            (p->id, p->instant, p->demand_cpu,
            p->demand_disk, p->demand_network)));

            // Incrementando o tamanho do vetor de eventos.
            size_events++;
        }
    } else {
        // Se existir processo em execucao na CPU, a demanda
        //de CPU do processo eh decrementada.
        pc->consume_cpu();

        // Caso a demanda de CPU do processo seja igual a 0,
        //o processo eh retirado da execucao da CPU.
        if (pc->running_cpu->demand_cpu == 0) {
            // Salvando o processo em execucao na CPU.
            process* p = pc->running_cpu;

            // Registrando o evento de espera do disco.
            events_vec[size_events] = (new event
            (time, WAIT_DISK, new process
            (p->id, p->instant, p->demand_cpu,
            p->demand_disk, p->demand_network)));

            // Incrementando o tamanho do vetor de eventos.
            size_events++;

            // Adiciona o processo na fila de espera de um dos discos.
            pc->add_process_disk();

            // Retirando o processo da execucao da CPU.
            pc->running_cpu = nullptr;

            p = pc->cpu->pop();

            if (p != nullptr) {
                // Se existir, o processo eh adicionado a execucao
                // da CPU.
                pc->running_cpu = p;

                // Registrando o evento de acesso a CPU.
                events_vec[size_events] = (new event
                (time, ACCESS_CPU, new process
                (p->id, p->instant, p->demand_cpu,
                p->demand_disk, p->demand_network)));

                // Incrementando o tamanho do vetor de eventos.
                size_events++;
            }
        }
    }

// PROCESS_DISK_1
// Verificando se nao existe processo em execucao no disco 1.
if (!pc->has_process_in_disk_1()) {
    // Retirando o processo da fila de espera do disco 1.
```

```
process* p = pc->disk_1->pop();

// Verificando se existe processo na fila de espera do disco 1.
if (p != nullptr) {
    // Adicionando o processo a execucao do disco 1.
     pc->running_disk_1 = p;

    // Registrando o evento de acesso ao disco.
     events_vec[size_events] = (new event
        (time, ACCESS_DISK, new process
        (p->id, p->instant, p->demand_cpu,
        p->demand_disk, p->demand_network)));

        // Incrementando o tamanho do vetor de eventos.
        size_events++;
    }
} else {
    // Se existir processo em execucao no disco 1,
    // a demanda de disco do processo eh decrementada.
    pc->running_disk_1->demand_disk--;
}

// PROCESS_DISK_2
// Verificando se nao existe processo em execucao no disco 2.
if (!pc->has_process_in_disk_2()) {
    // Retirando o processo da fila de espera do disco 2.
    process* p = pc->disk_2->pop();

    // Verificando se existe processo na fila de espera do disco 2.
    if (p != nullptr) {
        // Adicionando o processo a execucao do disco 2.
        pc->running_disk_2 = p;

        // Registrando o evento de acesso ao disco.
        events_vec[size_events] = (new event
        (time, ACCESS_DISK, new process
        (p->id, p->instant, p->demand_cpu,
        p->demand_disk, p->demand_network)));

        // Incrementando o tamanho do vetor de eventos.
        size_events++;
    }
} else {
    // Se existir processo em execucao no disco 2, a demanda
    // de disco do processo eh decrementada.
    pc->running_disk_2->demand_disk--;
}

// CONSUME_DISK
// Processo em execucao no disco 1.
process* disk1_p = pc->running_disk_1;

// Verificando se o processo em execucao no disco 1 terminou.
if (pc->has_process_in_disk_1() && disk1_p->demand_disk == 0) {
    // Salvando o processo em execucao no disco 1.
    process* p = disk1_p;
```

```
    // Retirando o processo da execucao do disco 1.
    pc->running_disk_1 = nullptr;

    // Registrando o evento de espera da rede.
    events_vec[size_events] = (new event
        (time, WAIT_NETWORK, new process
        (p->id, p->instant, p->demand_cpu,
        p->demand_disk, p->demand_network)));

    // Incrementando o tamanho do vetor de eventos.
    size_events++;

    // Adicionando o processo a fila de espera da rede.
    this->system->network->push(p);
}
// Processo em execucao no disco 2.
process* disk2_p = pc->running_disk_2;

// Verificando se o processo em execucao no disco 2 terminou.
if (pc->has_process_in_disk_2() && disk2_p->demand_disk == 0) {
    // Salvando o processo em execucao no disco 2.
    process* p = disk2_p;
    // Retirando o processo da execucao do disco 2.
    pc->running_disk_2 = nullptr;

    // Registrando o evento de espera da rede.
    events_vec[size_events] = (new event
         (time, WAIT_NETWORK, new process
         (p->id, p->instant, p->demand_cpu,
         p->demand_disk, p->demand_network)));

    // Incrementando o tamanho do vetor de eventos.
    size_events++;

    // Adicionando o processo a fila de espera da rede.
    this->system->network->push(p);
}

// PROCESS_NETWORK

// Verificando se nao existe processo em execucao na rede.
if (this->system->running_network == nullptr) {
    // Retirando o processo da fila de espera da rede.
    process* p = this->system->network->pop();

    // Verificando se existe processo na fila de espera
    // da rede.
    if (p != nullptr) {
        // Adicionando o processo a execucao da rede.
        this->system->running_network = p;

        // Registrando o evento de acesso a rede.
        events_vec[size_events] = (new event
            (time, ACCESS_NETWORK, new process
            (p->id, p->instant, p->demand_cpu,
            p->demand_disk, p->demand_network)));
```

```
                // Incrementando o tamanho do vetor de eventos.
                size_events++;
            }
        } else {
            // Se existir processo em execucao na rede, a demanda
            // de rede do processo eh decrementada.
            this->system->running_network->demand_network--;
        }

        // CONSUME_NETWORK
        // Verificando se o processo em execucao na rede terminou.
        if (this->system->running_network != nullptr &&
            this->system->running_network->demand_network == 0) {

        // Salvando o processo em execucao na rede.
        process* p = this->system->running_network;
        // Retirando o processo da execucao da rede.
        this->system->running_network = nullptr;

        // Registrando o evento de termino do processo.
        events_vec[size_events] = (new event
        (time, FINISH_PROCESS, new process
        (p->id, p->instant, p->demand_cpu,
        p->demand_disk, p->demand_network)));

        // Incrementando o tamanho do vetor de eventos.
        size_events++;

        // Deletando o processo. (Lembrando que os eventos guardam
        // uma copia do processo, entao nao ha problema em deletar
        // o processo aqui)
        delete p;
    }
}
```

- **2 Registro de eventos.**
  Após o escalonador terminar sua execução(o fim da execução do sistema), temos um registro de todos os eventos que aconteceram no sistema, o bloco de código a seguir é responsável por organizar todos esses eventos em uma matriz de eventos, e criar um objeto metrics para calcular as métricas.

```
// Cria uma fila de eventos para cada processo (8 eventos por processo)
for (int i = 0; i < (qtd_process * 8); i++) {
     events.push(events_vec[i]);
}

// Contador de eventos registrados para cada processo
int* counter_event = new int[qtd_process];

// Inicializando contadores
for (int i = 0; i < qtd_process; i++) {
        counter_event[i] = 0;
}

// Cada evento é colocado em sua devida linha de processo de acordo com
```

```
// o ID do processo, os eventos de cada processo serão colocados em ordem
// cronológica.
for (int i = 0; i < (qtd_process * 8); i++) {
    // Retirando o evento da fila de eventos.
    event* e = events.pop();

    // e->print();  // Usado para debug.
    // std::cout << std::endl;  // Usado para debug.

    // Registrando o evento na matriz de eventos.
    log[e->process_a->id][counter_event[e->process_a->id]] = e;
    // Incrementando o contador de eventos do processo.
    counter_event[e->process_a->id]++;
}

// Deletando o vetor do contador de eventos.
delete[] counter_event;

// Calculando as métricas e salvando em um objeto.
        metrics* m = new metrics(time - 1, qtd_process, log);

// Imprimindo as métricas.
m->print();

// Salvando as métricas em um arquivo.
m->save(filename, system->amount, system->policy == SJF ? "SJF" : "FCFS");
```

05. Testes Realizados

Podemos analisar a perfomance das políticas setando a semente de geração de números aleatórios para 123 para remover o fator aleatoriedade do sistema(Descomente a linha 61 da main.cpp).

Ao executarmos o programa usando o arquivo de processos arquivo_de_entrada_8.txt, com dois computadores obtemos os seguintes resultados em seguida obtemos:

Na política FCFS

- Tempo total de execução: 958
- Tempo médio de execução: 465.412
- Tempo médio de espera: 377.098
- Taxa de processamento: 0.05323591

Na política SJF

- Tempo total de execução: 960
- Tempo médio de execução: 402.882
- Tempo médio de espera: 314.569
- Taxa de processamento: 0.05312500

Para esse arquivo de entrada, podemos observar que a política FCFS teve um tempo total de execução menor(dois instantes de tempo), mas por outro lado o tempo médio de execução e o tempo médio de espera foram maiores. A taxa de processamento foi relativamente parecida.

Além de poder testar todos os arquivos, caso queira ver passo a passo de uma execução de processo, descomente a linha 309 e 310 da classe escalonator. Ao fazer isso o sistema vai printar o registro de eventos na ordem que foram executados.

Ao fazer o seguinte teste obtemos:

```
./main.exe FCFS teste01.txt 1
INITIALIZE_PROCESS
ID:             0
time:           0
inicio:         0
demand_cpu:     5
demand_disk:    4
demand_network: 3

WAIT_CPU
ID:             0
time:           0
inicio:         0
demand_cpu:     5
demand_disk:    4
demand_network: 3

INITIALIZE_PROCESS
ID:             1
time:           0
inicio:         0
demand_cpu:     5
demand_disk:    4
demand_network: 2

WAIT_CPU
ID:             1
time:           0
inicio:         0
demand_cpu:     5
demand_disk:    4
demand_network: 2

ACCESS_CPU
ID:             0
time:           0
inicio:         0
demand_cpu:     5
demand_disk:    4
demand_network: 3

INITIALIZE_PROCESS
ID:             2
time:           1
inicio:         1
demand_cpu:     5
demand_disk:    4
demand_network: 2

WAIT_CPU
ID:             2
time:           1
inicio:         1
demand_cpu:     5
demand_disk:    4
demand_network: 2
```

```
INITIALIZE_PROCESS
ID:             3
time:           2
inicio:         2
demand_cpu:     5
demand_disk:    4
demand_network: 2

WAIT_CPU
ID:             3
time:           2
inicio:         2
demand_cpu:     5
demand_disk:    4
demand_network: 2

WAIT_DISK
ID:             0
time:           5
inicio:         0
demand_cpu:     0
demand_disk:    4
demand_network: 3

ACCESS_CPU
ID:             1
time:           5
inicio:         0
demand_cpu:     5
demand_disk:    4
demand_network: 2

ACCESS_DISK
ID:             0
time:           5
inicio:         0
demand_cpu:     0
demand_disk:    4
demand_network: 3

WAIT_NETWORK
ID:             0
time:           9
inicio:         0
demand_cpu:     0
demand_disk:    0
demand_network: 3

ACCESS_NETWORK
ID:             0
time:           9
inicio:         0
demand_cpu:     0
demand_disk:    0
demand_network: 3
```

```
WAIT_DISK
ID:             1
time:           10
inicio:         0
demand_cpu:     0
demand_disk:    4
demand_network: 2

ACCESS_CPU
ID:             2
time:           10
inicio:         1
demand_cpu:     5
demand_disk:    4
demand_network: 2

ACCESS_DISK
ID:             1
time:           10
inicio:         0
demand_cpu:     0
demand_disk:    4
demand_network: 2

FINISH_PROCESS
ID:             0
time:           12
inicio:         0
demand_cpu:     0
demand_disk:    0
demand_network: 0

WAIT_NETWORK
ID:             1
time:           14
inicio:         0
demand_cpu:     0
demand_disk:    0
demand_network: 2

ACCESS_NETWORK
ID:             1
time:           14
inicio:         0
demand_cpu:     0
demand_disk:    0
demand_network: 2

WAIT_DISK
ID:             2
time:           15
inicio:         1
demand_cpu:     0
demand_disk:    4
demand_network: 2
```

```
ACCESS_CPU
ID:             3
time:           15
inicio:         2
demand_cpu:     5
demand_disk:    4
demand_network: 2

ACCESS_DISK
ID:             2
time:           15
inicio:         1
demand_cpu:     0
demand_disk:    4
demand_network: 2

FINISH_PROCESS
ID:             1
time:           16
inicio:         0
demand_cpu:     0
demand_disk:    0
demand_network: 0

WAIT_NETWORK
ID:             2
time:           19
inicio:         1
demand_cpu:     0
demand_disk:    0
demand_network: 2

ACCESS_NETWORK
ID:             2
time:           19
inicio:         1
demand_cpu:     0
demand_disk:    0
demand_network: 2

WAIT_DISK
ID:             3
time:           20
inicio:         2
demand_cpu:     0
demand_disk:    4
demand_network: 2

ACCESS_DISK
ID:             3
time:           20
inicio:         2
demand_cpu:     0
demand_disk:    4
demand_network: 2
```

```
FINISH_PROCESS
ID:             2
time:           21
inicio:         1
demand_cpu:     0
demand_disk:    0
demand_network: 0

WAIT_NETWORK
ID:             3
time:           24
inicio:         2
demand_cpu:     0
demand_disk:    0
demand_network: 2

ACCESS_NETWORK
ID:             3
time:           24
inicio:         2
demand_cpu:     0
demand_disk:    0
demand_network: 2

FINISH_PROCESS
ID:             3
time:           26
inicio:         2
demand_cpu:     0
demand_disk:    0
demand_network: 0

Tempo total de execução: 26
Tempo médio de execução: 18.000
Tempo médio de espera: 6.750
Taxa de processamento: 0.15384615
```

Os testes podem ser feitos com qualquer arquivo de texto com o formato certo salvo na pasta src/in, fique a vontade para testar as entradas que já estão lá.

06. Dificuldades encontradas

As maiores dificuldades encontradas foram:

- Compreender a lógica do escalonador, e como as classes iriam se relacionar, muito bem resolvidos com o modelo de classes que a Erica imaginou.
- Implementar o método run do sistema, ou seja, pensar na lógica que faria os processos serem executados e como o relógio lógico funcionaria.
- Forma que as métricas seriam calculadas, problema que foi facilmente resolvido com a criação de objetos feitos para salvar os eventos do sistema.

07. Conclusões

O algoritmo de escalonamento Shortest Job First (SJF) demonstrou ser mais eficiente do que o algoritmo First Come First Served (FCFS) com base nos testes realizados nos arquivos de entrada do simulador. Isso se deve a algumas razões:

- Priorização dos processos mais curtos: O SJF prioriza a execução dos processos mais curtos antes dos mais longos. Isso minimiza o tempo de espera, uma vez que os processos curtos são concluídos mais rapidamente, permitindo que outros processos sejam executados mais cedo.
- Redução do tempo de espera médio: Ao executar os processos mais curtos primeiro, o SJF reduz o tempo médio que os processos precisam esperar na fila antes de serem atendidos. Isso leva a um melhor desempenho geral do sistema, uma vez que os processos são concluídos de forma mais eficiente.
- Melhor utilização dos recursos: O SJF contribui para uma utilização mais eficiente da CPU, uma vez que os processos mais curtos são rapidamente liberados, permitindo que outros processos sejam escalonados. Isso evita o bloqueio prolongado de processos longos, o que pode causar gargalos no sistema.

Portanto, com base nos resultados dos testes realizados, podemos concluir que o SJF é uma escolha preferencial em relação ao FCFS quando se trata de escalonamento de processos, uma vez que ele melhora o desempenho do sistema, reduzindo o tempo de espera e otimizando a utilização dos recursos da CPU.