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

Francisco Djalma Pereira da Silva Júnior - 554222
Linyker Vinicius Gomes Barbosa - 556280

10 de novembro de 2023

1 Introdução

O projeto tem como objetivo realizar uma simulação para avaliar o desempenho de duas políticas de escalonamento, FCFS (First-Come-First-Served) e SJF (Shortest Job First), em um ambiente distribuído chamado NeoLook. Este sistema consiste em N computadores interligados por meio de uma rede de alta velocidade, sendo que cada computador possui uma CPU e dois discos. Ao submeter um processo para execução, ele passa por várias etapas, incluindo a fase de processamento na CPU, o acesso a um dos discos e a transferência de dados pela rede.

O propósito central da simulação é reproduzir a entrada de um número específico de processos no sistema. A partir dessa simulação, serão calculados indicadores como o tempo médio de execução dos processos, o tempo médio de espera e a taxa de processamento do sistema. Esses indicadores desempenharão um papel crucial na avaliação do desempenho do sistema sob as duas políticas de escalonamento mencionadas.

2 Estruturas de dados

2.1 Queue.h

Uma fila é uma estrutura de dados linear que segue a ordem "Primeiro a Entrar, Primeiro a Sair"(FIFO - First In, First Out). Ela mantém uma coleção de elementos onde a remoção de elementos ocorre no início da fila e a inserção de elementos ocorre no final da fila. No âmbito do projeto Neolook, a estrutura de dados TAD Queue será empregada na implementação do algoritmo FCFS, que adere ao princípio FIFO (First-In, First-Out).

2.2 PriorityQueue.h

Esta classe representa uma fila de prioridade baseada em uma min heap, sendo uma estrutura de dados que mantém uma coleção de elementos onde o elemento de menor valor (ou maior prioridade) está sempre no topo da fila. Ela é útil em situações em que se deseja garantir que o elemento de menor valor possa ser rapidamente recuperado e removido. A min heap é uma árvore binária completa em que cada nó pai tem um valor menor ou igual ao de seus nós filhos, garantindo que o elemento de menor valor esteja sempre na raiz.

A escolha da estrutura de dados TAD PriorityQueue no contexto do algoritmo SFJ (Shortest Job First) é justificada pela necessidade de ordenar a lista de processos com base no tempo de execução restante de cada processo, de forma a priorizar a execução dos mais curtos.

3 Divisão de Tarefas

Na distribuição de responsabilidades para este projeto, Linyker Vinícius foi designado para a implementação da estrutura de dados Queue, enquanto Djalma Júnior ficou encarregado da implementação da PriorityQueue. Trabalhando em estreita colaboração, a dupla construiu a lógica abrangente do sistema distribuído, incluindo a integração das filas no contexto do sistema NeoLook, abordando aspectos como processamento na CPU, acesso a discos e transferência de dados pela rede.

4 Implementação

As classes serão apresentadas à medida que estabelecermos essas conexões, iniciando com uma explicação básica de cada classe. As classes implementadas no sistema incluem:

4.1 Process

O arquivo Process.h define a estrutura de dados Process, que representa um processo no contexto do sistema. Cada instância da estrutura possui atributos como identificador, instante de chegada, demanda de execução na CPU, demanda de execução no disco e demanda de execução na rede.

```
struct Process {
    unsigned id{};               // identificador do processo
    unsigned instant{};          // instante de chegada do processo
    unsigned short d_cpu{};      // demanda de execução na CPU
    unsigned short d_disk{};     // demanda de execução no disco
    unsigned short d_network{};  // demanda de execução na rede

    Process(const unsigned& id,
            const unsigned& instant,
            const unsigned short& d_cpu,
            const unsigned short& d_disk,
            const unsigned short& d_network);

    Process();

    ~Process();
};
```

4.2 Resource

A classe Resource é uma abstração que representa um recurso genérico no sistema, como uma CPU, disco ou rede. A classe contém atributos e métodos que são comuns a todos esses tipos de recursos, permitindo uma implementação modular e reutilizável.

A classe Resource desempenha um papel crítico no sistema, gerenciando a execução dos processos nas diferentes partes do sistema (CPU, disco, rede). Sua modularidade e flexibilidade são essenciais para a implementação de diferentes políticas de escalonamento e comportamentos específicos de cada recurso. Além disso, a classe mantém registros detalhados das operações no arquivo de log, facilitando o monitoramento e análise do desempenho do sistema.

```
class Resource {
   protected:
    Process* process;             // processo que está sendo executado na CPU
    Queue<Process*>* queue;       // fila de processos da CPU
    PriorityQueue<Process*>* pq;  // fila de processos da CPU
    bool politica;                // politica de escalonamento. 0 - FCFS, 1 - SJF
    bool busy;                    // indica se a CPU está ocupada
    LogFile* logFile;             // arquivo de log
```

```
 unsigned short counter;       // contador de tempo

private:
public:
 /**
  * @brief Construct a new Resource object
  *
  */
 Resource();

 /**
  * @brief Construct a new Resource object
  *
  * @param politica
  * @param logFile
  */
 Resource(bool politica, LogFile* logFile);

 /**
  * @brief Destroy the Resource object
  *
  */
 virtual ~Resource();

 /**
  * @brief Verifica se a CPU está ocupada
  *
  * @return true
  * @return false
  */
 bool isBusy() const;

 /**
  * @brief Retorna o processo que está sendo executado na CPU
  *
  * @return Process*
  */
 Process* getProcess() const;

 /**
  * @brief Carrega um processo na CPU da fila de processos
  *
  */
 virtual void loadFromQueue();

 /**
  * @brief Executa o processo que está na CPU em um ciclo
  *
  */
 virtual void execute();

 /**
  * @brief Verifica se o processo que está na CPU terminou sua execução
  *
  * @return true
  * @return false
```

```
     */
    virtual bool isFinished();

    /**
     * @brief Remove o processo que está na CPU
     *
     */
    virtual void finishProcess();

    /**
     * @brief Escolhe se o processo será executado na CPU ou inserido na fila de processos
     *
     * @param p
     */
    virtual void setProcess(Process* p) = 0;
};

class CPU : public Resource {
   public:
    CPU();

    CPU(bool politica, LogFile* logFile);

    ~CPU();

    void setProcess(Process* p) override;
};

class Disk : public Resource {
   public:
    Disk();

    Disk(bool politica, LogFile* logFile);

    ~Disk();

    void setProcess(Process* p) override;
};

class Network : public Resource {
   public:
    Network();

    Network(bool politica, LogFile* logFile);

    ~Network();

    void setProcess(Process* p) override;
};
```

4.3 Computer

A classe Computer representa um computador no sistema, composto por uma CPU e dois discos. A classe é responsável por gerenciar esses recursos e fornecer métodos para acessá-los. Desempenha um papel crucial no sistema, representando a infraestrutura principal onde os processos são executados. Sua composição de recursos, como CPU, discos e rede, é essencial para a execução e coordenação

eficiente dos processos no ambiente computacional. A classe fornece métodos para acessar e interagir com esses recursos, facilitando a implementação do escalonamento e a execução dos processos.

```
class Computer {
   private:
    CPU* cpu;          // CPU do computador
    Disk* disk;        // Discos do computador
    bool politica;     // politica de escalonamento da CPU. 0 - FCFS, 1 - SJF
    Network* network;  // rede do computador

   public:
    /**
     * @brief Construtor padrão da classe Computer.
     *
     */
    Computer();

    /**
     * @brief Construtor parametrizado da classe Computer.
     *
     * @param politica Política de escalonamento da CPU. 0 - FCFS, 1 - SJF
     * @param net Ponteiro para a rede do computador.
     */
    Computer(bool politica, Network* net, LogFile* logFile);

    /**
     * @brief Destrutor da classe Computer.
     *
     * Libera a memória alocada para o array de disco.
     *
     */
    ~Computer();

    /**
     * @brief Retorna a CPU do computador.
     *
     * @return Referência para a CPU do computador.
     */
    CPU& getCPU();

    /**
     * @brief Retorna uma referência para o objeto Disk no índice especificado.
     *
     * @param i O índice do objeto Disk desejado.
     * @return Uma referência para o objeto Disk no índice especificado.
     */
    Disk& getDisk(int i);
};

#endif
```

4.4 LogFile

A classe LogFile representa o arquivo de log do sistema, responsável por registrar informações importantes sobre a execução dos processos e o desempenho do sistema. Desempenha um papel vital no sistema, registrando eventos cruciais durante a simulação. Ela fornece uma visão detalhada das

operações, facilitando a análise do desempenho do sistema e a depuração de possíveis problemas. Os métodos específicos para diferentes eventos contribuem para um registro organizado e compreensível das atividades do sistema.

```
 class LogFile {
private:
 std::ofstream* m_file;  // ponteiro para o arquivo de log
 unsigned* timer;        // tempo atual do sistema
 bool detailed{};        // indica se o log é detalhado

public:
 /**
  * @brief Construtor padrão da classe LogFile.
  * Chama a função createLogArchiveName() para criar o nome do arquivo de log.
  * Cria o arquivo de log.
  *
  */
 LogFile(unsigned* time, bool detailed);

 /**
  * @brief Destrutor da classe LogFile.
  *
  */
 ~LogFile();

 /**
  * @brief Cria o nome do arquivo de log. O nome é composto
  pela data e hora da criação do arquivo.
  *
  * @return Uma string com o nome do arquivo de log.
  */
 std::string createLogArchiveName();

 /**
  * @brief Função que verifica se o arquivo de log está aberto.
  *
  * @return True, se o arquivo estiver aberto. False, caso contrário.
  */
 bool isOpen();

 // Messages for CPU ------------------------------------------------------

 /**
  * @brief Cria uma mensagem de carregamento direto do processo na CPU.
  *
  * @param time Tempo atual da simulação.
  * @param processId Identificador do processo.
  * @param computerId Identificador do computador.
  */
 void executionCPU(const unsigned& processId);

 /**
  * @brief Criar uma mensagem de finalização de execução do processo na CPU.
  *
  * @param time Tempo atual da simulação.
  * @param processId Identificador do processo.
  * @param computerId Identificador do computador.
```

```
 */
void executionCompletedCPU(const unsigned& processId);

// Messages for Disk ---------------------------------------------------

/**
 * @brief Cria uma mensagem de carregamento direto do processo no disco.
 *
 * @param time Tempo atual da simulação.
 * @param processId Identificador do processo.
 * @param diskId Identificador do disco.
 * @param computerId Identificador do computador.
 */
void executionDisk(const unsigned& processId);

/**
 * @brief Cria uma mensagem de carregamento do processo na fila de espera do disco.
 *
 * @param time Tempo atual da simulação.
 * @param processId Identificador do processo.
 * @param diskId Identificador do disco.
 * @param computerId Identificador do computador.
 */
void loadedIntoQueue(const unsigned& processId);

/**
 * @brief Cria uma mensagem de finalização de execução do processo no disco.
 *
 * @param time Tempo atual da simulação.
 * @param processId Identificador do processo.
 * @param diskId Identificador do disco.
 * @param computerId Identificador do computador.
 */
void executionCompletedDisk(const unsigned& processId);

// Messages for Network ------------------------------------------------

/**
 * @brief Cria uma mensagem de carregamento direto do processo na rede.
 *
 * @param time Tempo atual da simulação.
 * @param processId Identificador do processo.
 */
void executionNetwork(const unsigned& processId);

/**
 * @brief Cria uma mensagem de finalização de execução do processo na rede.
 *
 * @param time Tempo atual da simulação.
 * @param processId Identificador do processo.
 */
void executionCompletedNetwork(const unsigned& processId);

/**
 * @brief Cria uma mensagem de carregamento do processo
 da fila de espera para execução na rede.
```

```
     *
     * @param time Tempo atual da simulação.
     * @param processId Identificador do processo.
     */
    void loadedFromQueue(const unsigned& processId);

    // Messages for Process ------------------------------------------------

    /**
     * @brief Cria uma mensagem de criação do processo.
     *
     * @param time Tempo atual da simulação.
     * @param processId Identificador do processo.
     */
    void processCreated(const unsigned& processId);

    /**
     * @brief Cria uma mensagem de finalização do processo.
     *
     * @param time Tempo atual da simulação.
     * @param processId Identificador do processo.
     */
    void processFinished(const unsigned& processId);

    // Standard messages ---------------------------------------------------

    /**
     * @brief Cria o cabeçalho do arquivo de log.
     *
     * @param qtdProcess Quantidade de processos.
     * @param politica Política de escalonamento.
     * @param qtdPcs Quantidade de computadores.
     */
    void headerMessage(const unsigned& qtdProcess,
                       const bool& politica,
                       const unsigned short& qtdPcs);

    /**
     * @brief Cria a mensagem de conclusão da simulação.
     *
     */
    void executionCompleted();

    /**
     * @brief Cria a mensagem com as estatísticas da simulação.
     *
     * @param timer Tempo total de execução da simulação.
     * @param averageExecution Tempo médio de execução dos processos.
     * @param averageWaiting Tempo médio de espera dos processos.
     * @param processingFee Taxa de processamento.
     */
    void statistics(unsigned long timer, double averageExecution,
                    double averageWaiting, double processingFee);
};
```

4.5 System

A classe System é o cerne da simulação do sistema operacional, representando a coordenação entre todos os componentes do sistema. Sua principal função é orquestrar o fluxo de processos, computadores, discos e a rede. Durante a execução, ela gerencia o envio de processos para a CPU, a execução nos computadores, o redirecionamento para discos e a transmissão pela rede. Ao final da simulação, a classe calcula estatísticas cruciais, como o tempo médio de espera e execução, além da taxa de processamento. A implementação adequada dessa classe é fundamental para garantir uma simulação precisa e útil na avaliação do desempenho do sistema operacional.

No âmbito específico do código fornecido, a classe System abriga métodos para carregar um arquivo de trace, que contém informações sobre a chegada e demandas de processos, e executa a simulação, coordenando a interação entre os diversos recursos computacionais. Além disso, ela gerencia a criação e o encerramento de processos, monitorando o tempo de execução e alocando-os aos recursos apropriados, como CPU, discos e rede, de acordo com a política de escalonamento estabelecida.

```
void System::execute() {
    unsigned pendentes = qtdProcessos;
    // std::srand(time(NULL));
    std::srand(123);
    Vector<Process>::iterator it = processos->begin();
    Vector<Process>::iterator end = processos->end();
    logFile->headerMessage(pendentes, politica, qPcs);

    while (true) {
        // CARREGA OS PROCESSOS NA CPU ----------------------------
        while (it != end && it->instant == timer) {
            int pc = rand() % qPcs;
            computers[pc]->getCPU().setProcess(&(*it));
            it++;
        }

        for (unsigned i = 0; i < qPcs; i++) {
            // CPU ------------------------------------------------
            CPU& cpu = computers[i]->getCPU();
            if (cpu.isFinished()) {
                Process* p = cpu.getProcess();
                cpu.finishProcess();
                int disk = rand() % 2;
                computers[i]->getDisk(disk).setProcess(p);
                cpu.loadFromQueue();
            }
            cpu.execute();
            // DISCOS  --------------------------------------------
            for (unsigned j = 0; j < 2; j++) {
                Disk& disk = computers[i]->getDisk(j);
                if (disk.isFinished()) {
                    Process* p = disk.getProcess();
                    disk.finishProcess();
                    network->setProcess(p);
                    disk.loadFromQueue();
                }
                disk.execute();
            }
        }

        // REDE -----------------------------------------------------
        if (network->isFinished()) {
```

```
            Process* p = network->getProcess();
            network->finishProcess();
            unsigned tempoExecucao = timer - p->instant;
            unsigned tempoEspera = tempoExecucao - (p->d_cpu + p->d_disk + p->d_network);
            tempoMedioExecucao += (double)tempoExecucao / qtdProcessos;
            tempoMedioEspera += (double)tempoEspera / qtdProcessos;
            network->loadFromQueue();
            pendentes--;
            if (pendentes == 0) break;
        }
        network->execute();

        timer++;  // incrementa o relógio lógico
    }
    taxaProcessamento = (double)qtdProcessos / (timer - processos->begin()->instant);
    std::cout << "Política de escalonamento = " << (politica ? "SJF" : "FCFS") << std::endl;
    std::cout << "Quantidade de processos   = " << qtdProcessos << std::endl;
    std::cout << "Tempo total de execução   = " << timer << std::endl;
    std::cout << std::fixed << std::setprecision(2) << "Tempo médio de espera     = "
    << tempoMedioEspera << std::endl;
    std::cout << std::fixed << std::setprecision(2) << "Tempo médio de execução   = "
    << tempoMedioExecucao << std::endl;
    std::cout << std::fixed << std::setprecision(6) << "Taxa de processamento     = "
    << taxaProcessamento << std::endl
              << std::endl;
    logFile->executionCompleted();
    logFile->statistics(timer, tempoMedioExecucao, tempoMedioEspera, taxaProcessamento);
}
```

5 Testes Realizados

Para a semente (seed) 123 e um único computador, foram realizadas simulações utilizando duas políticas de escalonamento distintas: SJF (Shortest Job First) e FCFS (First-Come, First-Served). Os resultados obtidos são apresentados abaixo para uma quantidade de 1001 processos:

5.1 Política SJF

- Tempo total de execução: 24876 unidades de tempo.
- Tempo médio de espera: 9756.09 unidades de tempo.
- Tempo médio de execução: 9839.71 unidades de tempo.
- Taxa de processamento: 0.040240 processos por unidade de tempo.

5.2 Política FCFS

- Tempo total de execução: 25251 unidades de tempo.
- Tempo médio de espera: 12528.32 unidades de tempo.
- Tempo médio de execução: 12611.94 unidades de tempo.
- Taxa de processamento: 0.039642 processos por unidade de tempo.

Comparando os resultados, observa-se que a política SJF apresentou um tempo total de execução menor e uma taxa de processamento ligeiramente superior em relação à política FCFS. No entanto, a política FCFS resultou em tempos médios de espera e de execução um pouco mais altos. A escolha entre essas políticas dependerá das prioridades e requisitos específicos do sistema em questão.

6 Dificuldades Encontradas

A otimização do código se mostrou um desafio significativo. A busca por um equilíbrio entre desempenho e legibilidade foi constante, exigindo constantes revisões para aprimorar a eficiência sem sacrificar a clareza do código-fonte.

Outra dificuldade enfrentada foi lidar com entradas muito longas, o que exigiu estratégias especiais para gerenciar grandes volumes de dados sem comprometer o desempenho do sistema. O design robusto e a implementação eficiente foram cruciais para lidar com essa demanda.

A adoção de programação orientada a objetos também representou um desafio, especialmente na correta estruturação das classes e na definição adequada de relações entre objetos. Superar essas dificuldades foi fundamental para assegurar uma arquitetura sólida e de fácil manutenção, contribuindo para a escalabilidade e flexibilidade do sistema.

7 Conclusões

O desenvolvimento deste sistema de simulação proporcionou uma experiência valiosa, evidenciando a importância de uma abordagem cuidadosa na implementação de TADs. A compreensão aprofundada dessas estruturas foi essencial para o sucesso do projeto, destacando a relevância de uma sólida fundação teórica para traduzir conceitos complexos em código funcional.

8 Links

Repositório no GitHub