UNIVERSIDADE FEDERAL DO CEARÁ - CAMPUS QUIXADÁ

ESTRUTURA DE DADOS - TURMA 02A

CIÊNCIA DA COMPUTAÇÃO

Prof. Atílio Gomes Luiz

Wesley Freitas Sobrinho e Pedro Arthur Brandão da Guia

PROJETO DE ESTRUTURA DE DADOS

Sistema de Escalonamento Neolook

QUIXADÁ

2023

1 INTRODUÇÃO

. O projeto proposto tem como objetivo simular duas políticas de escalonamento de processos diferentes (First come - first served e Shortest job first) em um sistema distribuído chamado Neolook. O sistema é composto por uma quantidade n de computadores, na qual os processos entram em um computador de forma aleatória, cada computador com uma CPU e dois discos, com uma rede compartilhada para todos os processos, cada processo tem uma demanda para cada recurso (CPU, disco e rede) e devem ser escalonados de acordo com a política, utilizando-se de filas implementadas por meio de TAD para guardar os processos à medida que chegam no recurso e a demanda do processo atual ainda não foi atendida, os processos passam para o próximo recurso a medida que sua execução no recurso em que estão atualmente termina, e assim segue até que todos os processos sejam executados. A sequência de recursos a ser seguida é CPU -> disco (o processo vai para um dos dois discos de forma aleatória) -> rede. Na entrada da simulação temos: o nome do programa, a política de escalonamento escolhida e um arquivo de texto contendo os processos que serão executados, (cada um com um instante de entrada, demanda de CPU, demanda de disco, e demanda de rede). As entradas devem ser fornecidas no terminal, e as saídas desejadas são o tempo médio de execução dos processos, o tempo médio de espera, e a taxa de processamento. A solução foi apresentada em código feito na linguagem c++, utilizando TAD e orientação a objetos, bem como alocação dinâmica de memória e estruturas de dados, com o objetivo de simular o NeoLook e produzir as saídas esperadas, para depois comparar o desempenho das duas políticas.

2 EXPLICAÇÃO DA ESTRUTURA DE DADOS

A estrutura de dados utilizada no projeto foi a fila, uma fila é uma estrutura que organiza elementos em uma ordem específica, seguindo o princípio do FCFS, uma das políticas propostas no projeto, significando que o primeiro elemento a entrar, é o primeiro elemento a sair. A fila tem como principal funcionalidade gerenciar itens de maneira sequencial, permitindo a adição de novos elementos no

final da fila, e a remoção de elementos do início da fila. A fila é comumente usada em situações em que a ordem de processamento dos elementos é crucial. Algumas de suas principais operações são: enfileirar, desenfileirar, acessar o primeiro elemento, acessar o último elementos e obter o tamanho. No projeto realizado, foram utilizados várias filas, como um fila para guardar os computadores(computers), uma fila para guardar os processos da entrada (arrivalQueue) e as filas de espera de cada recurso. As filas foram implementadas do zero, estando representada no arquivo Queue.h do código fornecido.

As complexidades das funções principais são: Todas as funções principais apresentam complexidade O(1), a única que apresenta complexidade O(n) é a queueState(), que mostra o estado atual da fila.

3 DIVISÃO DE TAREFAS

A execução do projeto foi feita em conjunto com ambos contribuindo para a construção do código. O esqueleto do código principal inicialmente foi feito de pelos dois, com o escolhido para implementação final sendo o de Wesley, onde ambos foram corrigindo os erros e adicionando o que precisava ser acrescentado. O relatório foi escrito por Pedro Arthur, e a lógica da saída foi feita de forma individual, com Wesley fazendo tempo médio de execução e tempo de espera, e Pedro Arthur implementando a lógica da taxa de processamento, o restante foi feito em conjunto, de forma presencial.

4 EXPLICAÇÃO DA IMPLEMENTAÇÃO

O projeto foi realizado na linguagem c++ e foi dividido em sete arquivos diferentes, sendo cinco de cabeçalho. Os arquivos são:

Queue.h: esse arquivo contém a implementação da fila feita do zero, a qual serão criadas as filas de cada recurso a partir dessa implementação. Contém

uma struct que representa os nós, uma classe para o iterador, e uma classe que representa a Queue criada e suas funções, a qual estão descritas no código.

Computer.h: contém uma classe computer, classe essa que possui um ponteiro para cpu, disco 1, disco 2 e para a rede compartilhada entre todos os computadores.

```
#ifndef COMPUTER_H
#define COMPUTER_H
#include "Resource.h"

class Computer
{
private:
    Resource *cpu;
    Resource *disk1;
    Resource *disk2;
    static Resource *network;

public:
    Computer();
    ~Computer();
    Resource &getCPU() const;
    Resource &getDisk1() const;
    Resource &getDisk2() const;
    Resource &getNetwork() const;
};

#endif
```

Computer.cpp: Apresenta a implementação dos métodos descritos em computer.h, com construtor e destrutor

Resource.h: Esse arquivo contém a classe resource, que representa o recurso de fato, com seus atributos, tendo uma variável booleana para representar se está disponível ou não, um ponteiro para o processo atual e a fila de espera do recurso. O método isAvailable retorna se o recurso está disponível ou não. O método Allocate() recebe um processo e o seta como atual, e consequentemente atualiza available para falso. O método release() libera o processo e consequentemente seta

available para verdadeiro. O método getCurrentProcess() retorna o processo atual, e getQueue() retorna a fila de espera do processo:

```
#ifndef RESOURCE_H
#define RESOURCE_H
#include "Queue.h"
#include "Process.h"

class Resource
{
private:
    bool available = true;
    Process *currentProcess = nullptr;
    Queue<Process *> waitingQueue{};
public:
    Resource() = default;

    bool isAvailable() const
    {
        return available;
    }
    // Aloca o recurso ao processo
    void allocate(Process *process)
    {
        this->currentProcess = process;
        this->available = false;
    }
    // libera o recurso
    void release()
    {
        this->currentProcess = nullptr;
        this->available = true;
    }
    // retorna o processo que está alocado no recurso
    Process *getCurrentProcess() const
    {
        return currentProcess;
    }

    // retorna a fila de espera do recurso
    Queue<Process *> &getQueue()
```

```
        {
            return waitingQueue;
        }
    };
    #endif
```

Process.h: Esse arquivo contém a classe processo, cada um com seu instante, e demanda dos recursos requeridos como atributo. Existem também outros 2 atributos: runtime, que guarda o tempo de execução, que será utilizado na produção das saídas do programa, e o waitingTime, que guarda o tempo que cada processo ficou na fila de espera, que também será importante na produção das estatísticas.

Simulator.h: Esse arquivo apresenta toda a lógica da simulação que foi realizada no projeto, sendo chamado na main através do método simulate() e realizando as operações necessárias para todos os recursos, chamando os getters e setters dos outros arquivos quando necessário, e retornando os valores finais quando a execução termina (quando não há mais processos ativos).
As complexidades das funções do Simulator são:

-Construtor de simulator: complexidade O(n)

-Destrutor de simulator: complexidade O(n)

-void simulate(): complexidade O(T * n), onde n é a quantidade de computadores e T é o tempo total de execução dos processos.

-int randomNumberGenerator(...): complexidade O(1)

-Process *updateCurrentProcess(...): complexidade O(1)

-void moveToNextResource(...): complexidade O(1)

-void escalation(...): Quando type é FCFS a complexidade é O(1). Quando type é SJF, no pior caso, a complexidade é O(n * log(n))

-void sortProcessQueue(...): complexidade O(n * log(n))

-double averageExecutionTime(): complexidade O(n)

-double averageWaitingTime(): complexidade O(n)

-double processingRate(): complexidade O(n)

Main.cpp: Apresenta a função principal do programa, que é chamada quando os dados são lidos do terminal.

Como foi feita a ordenação para SJF:

Foi implementado o método sortProcessQueue, que é responsável por ordenar a fila de espera de um recurso (cpu, disco ou rede). A ordenação é baseada em duas condições:

Menor Demanda: Primeiro, os processos são ordenados com base em suas demandas. Um processo com uma demanda menor deve ser executado antes de um processo com uma demanda maior.

Menor Instante: Se houver empate nas demandas, então a decisão é tomada com base no instante em que esses processos chegaram à fila de espera. O processo que chegou primeiro é escolhido para execução.

Explicação passo a passo:

Passo 1: Verificação da Quantidade de Elementos na Fila

O método começa verificando se há apenas um ou nenhum elemento na fila. Se houver apenas um ou nenhum elemento, a fila já está ordenada e não há necessidade de realizar mais operações de ordenação.

Passo 2: Inicialização do Nó Atual (`current`)

O método inicializa um ponteiro para o primeiro nó da fila. Esse ponteiro é usado para iterar sobre a fila de processos.

Passo 3: Loop Externo para Iterar sobre a Fila

Dentro deste loop, o algoritmo começa a percorrer a fila de processos com o objetivo de encontrar o processo com a menor demanda na fila de espera.

Passo 4: Inicialização do Nó Mínimo (`minNode`)

Para iniciar a comparação, o algoritmo assume que o primeiro nó é o que contém o processo de menor demanda até o momento.

Passo 5: Loop Interno para Comparar Processos

Dentro deste loop interno, o algoritmo compara o processo no nó atual (`current`) com os processos nos nós subsequentes na fila (`nextNode`). A comparação é feita com base na demanda dos processos e no instante em que chegaram à fila de espera.

- Se o processo em `nextNode` tiver uma demanda menor do que o processo em `minNode`, ou se tiverem a mesma demanda mas chegaram à fila antes, então `minNode` é atualizado para apontar para `nextNode`.

- O loop interno continua até que todos os nós na fila tenham sido comparados com `minNode`.

Passo 6: Troca de Elementos

Após encontrar o nó com o processo de menor demanda na fila, o algoritmo troca os elementos nos nós `current` e `minNode`. Isso coloca o processo de menor demanda na posição correta na fila de espera.

Passo 7: Iteração e Continuação do Loop Externo

Após a troca, o algoritmo move o ponteiro `current` para o próximo nó na fila e continua o loop externo. Esse processo de encontrar o mínimo e trocar os elementos é repetido até que toda a fila esteja ordenada.

Passo 8: Conclusão

Quando o loop externo termina, a fila de processos foi ordenada com base em suas demandas, com os processos de menor demanda e menor instante de chegada à fila de espera aparecendo antes na fila.

5 MANUAL PARA EXECUTAR O SISTEMA

Primeiramente, digite no seu terminal:

./main.exe <política_de_escalonamento(FCFS ou SJF)> <arquivo_de_entrada.txt>

Posteriormente, o programa pedirá para que você digite a quantidade de computadores a serem utilizados na simulação.

Após digitar, é iniciado a simulação. Depois de terminar a simulação, será exibido três estatísticas referentes ao desempenho. As estatísticas são: Tempo médio de execução de cada processo, tempo de espera médio de cada processo e taxa de processamento.

6 TESTES REALIZADOS

°Com 5 computadores: (arquivo_de_texto_8.txt)
Para FCFS:
tempo médio de execução: 404.60784 segundos
tempo de espera médio: 318.29412 segundos
taxa de processamento: 0.0060827 segundos

Para SJF:
tempo médio de execução: 344.90126 segundos
tempo de espera médio: 256.58824 segundos
taxa de processamento: 0.0060827 segundos

°Com 10 computadores: (arquivo_de_entrada_5.txt)
Para FCFS:
tempo médio de execução: 7181.52747 segundos
tempo de espera médio: 7097.90509 segundos
taxa de processamento: 0.0006793 segundos

Para SJF:
tempo médio de execução: 4895.41858 segundos
tempo de espera médio: 4811.79620 segundos
taxa de processamento: 0.0006793 segundos

Considerações: comparando os testes feitos, podemos perceber que o SJF foi mais eficiente nos resultados obtidos, também foi observado que quando aumentamos a quantidade de computadores o tempo de processamento no simulador aumentou.

7 DIFICULDADES ENCONTRADAS

Enfrentamos desafios na ordenação da fila e na implementação da rede compartilhada. Além disso, observamos que nosso programa apresentou ineficiências ao lidar com entradas extensas, como no arquivo_de_entrada_4.txt, especialmente à medida que aumentamos o número de computadores.

8 CONCLUSÕES

Ao longo deste projeto, exploramos e implementámos um simulador para o sistema NeoLook, focando no estudo de duas políticas de escalonamento. A análise comparativa entre as políticas First Come, First Served (FCFS) e Shortest Job First (SJF) proporcionou insights valiosos sobre o desempenho relativo dessas abordagens.

A estrutura de dados fila revelou-se fundamental para a modelagem eficaz do gerenciamento de recursos no NeoLook. Através das operações de enfileiramento e desenfileiramento, pudemos simular o fluxo de processos, avaliando o tempo médio de execução, tempo médio de espera e taxa de processamento.

Esse projeto não apenas aprimorou nossas habilidades em programação c++, mas também proporcionou uma compreensão mais profunda dos desafios associados ao desenvolvimento de programas mais complexos.

9 REFERÊNCIAS UTILIZADAS

https://www.youtube.com/live/qveu-Iv6Tpk?si=-6xivWqSC9c4oGpB

https://youtu.be/Tp7yMXPc3b8?si=TPr3awHwVmWPYyQV

https://www.youtube.com/live/napD2SXz4Fo?si=Kooo--18eB6eKLV7

https://youtu.be/LNV1mig7nVg

https://youtu.be/BqmGWsnuHP0?si=Hc6-TIJLw5ej04nE

https://youtu.be/dxBb1rDunKU?si=d74mvqc2GPuMXz20