PCS3732 - Laboratório de Processadores

Simulador e Analisador de Algoritmos de Escalonamento de Processos

Breno Suguru Costa Tominaga-11804320
Sander Söhngen-11819873
João Pedro Aras-11803545

14 de Agosto de 2023

Sumário

Prof. Bruno Basseto

1 Introdução

Os algoritmos de escalonamento de processos têm um papel central no funcionamento dos sistemas operacionais, visto que são eles que controlam como os processos são executados na CPU. Essa função, apesar de complexa, é vital para otimizar o uso dos recursos computacionais, como a memória e os dispositivos de entrada/saída, e é por meio dela que métodos como o First-Come-First-Serve (FCFS), Round Robin, e Shortest Job First (SJF) ganham vida. No entanto, a compreensão dos algoritmos de escalonamento pode ser desafiadora, especialmente para aqueles que estão começando a estudar ciência da computação. É nesse contexto que a aplicação didática que simula o funcionamento desses algoritmos se torna tão importante. Ao proporcionar uma representação visual e interativa, essa ferramenta pedagógica permite que estudantes e profissionais possam realmente ver e entender como diferentes algoritmos afetam a eficiência do sistema. Assim, não apenas torna a aprendizagem mais acessível e envolvente, mas também ilustra de maneira clara e eficaz um conceito que, de outra forma, poderia permanecer abstrato e inacessível. Considerando isso, o propósito desse projeto consiste na simplificação da aprendizagem desses modelos de escalonamento através da criação de um simulador e analisador de algoritmos de escalonamento de processos.

2 Algoritmos de escalonamento de processos

Considerando o escopo e o objetivo do simulador, vão ser analisados os seguinte algoritmos de processos:

1. **First-Come-First-Serve (FCFS)**:
   O FCFS é o algoritmo de escalonamento mais simples e direto. Nele, os processos são executados na ordem em que chegam à fila de espera. O primeiro processo que chega é o primeiro a ser executado, e assim por diante. Embora seja fácil de implementar, o FCFS pode levar ao que é conhecido como "efeito de convoy", onde processos curtos precisam esperar um processo longo ser concluído, o que pode levar a uma ineficiência.

2. **Round Robin (RR)**:
   O Round Robin é um algoritmo de escalonamento preemptivo que distribui o tempo da CPU igualmente entre todos os processos em quantias de tempo fixas, chamadas de "quantum". Quando um processo é executado pelo tempo de um quantum, ele é colocado no final da fila, e o próximo processo da fila é executado. Isso garante que cada processo receba uma fatia igual de tempo, promovendo uma distribuição justa de recursos.

3. **Shortest Job First (SJF)**:
   Este algoritmo seleciona o processo com o menor tempo de execução estimado a seguir. Ao fazer isso, ele minimiza o tempo médio de espera, mas pode ser difícil implementar na prática, pois requer conhecimento prévio do tempo de execução de cada processo. Além disso, se processos mais longos continuarem chegando, eles podem ser indefinidamente adiados, um problema conhecido como inanição.

4. **Priority Round Robin (PriorityRR)**:
   O PriorityRR é uma variação do Round Robin que leva em consideração as prioridades dos processos. Neste algoritmo, cada processo é atribuído a uma prioridade, e o escalonamento é feito com base nessas prioridades. Processos com prioridades mais altas são executados antes daqueles com prioridades mais baixas. Dentro da mesma prioridade, os processos são escalonados usando o método Round Robin. Isso permite uma maior flexibilidade e pode ser útil em sistemas onde certos processos são mais críticos do que outros.

Vale observar, que cada um desses algoritmos tem suas próprias vantagens e desvantagens e é mais adequado para diferentes tipos de aplicativos e contextos. A escolha do algoritmo de escalonamento certo pode ter um impacto significativo no desempenho e na eficiência do sistema operacional.

3 Código explicado

Em primeiro lugar, é importante notar que a abordagem utilizada foi de desacoplamento entre a UI e o "backend"de forma a simular a separação que há entre o Software e Hardware em que o SO atua como intermediário, evidenciado pelo único código que o HTML tem acesso:

```

import InterfaceManager from "./interfaceManager.js";

document.addEventListener("DOMContentLoaded", () => {
    const interfaceManager = new InterfaceManager();
    interfaceManager.init();
  });

```

Em segundo lugar, é fundamental entender a estrutura utilizada e o papel de cada classe:

```

// Classe responsável por gerenciar a interface do usuário e o resto do código
class InterfaceManager

// Classes que implementam os algoritmos escolhidos
class Scheduler
class FCFS extends Scheduler
class SJF extends FCFS
class RR extends Scheduler
class PriorityRR extends Scheduler

//Classe que representa, de fato, o processo, sendo encapsulada por uma fábrica de processos:
class Process

static class ProcessFactory
    static createProcessWithValues(executionTime, priority)
    static createCopiedProcess(process)
```

Por fim, vamos aprofundar sobre métodos importantes de cada uma dessas classes:
Começando pelos schedulers:

```

export class FCFS extends Scheduler {
  async run(interfaceManager) {
    while (this.processQueue.length > 0) {
      interfaceManager.updateVisualQueue(this.processQueue);
      const process = this.processQueue.shift();

      this.totalExecutionTime += process.remainingTime;
      await process.execute(process.remainingTime, interfaceManager);

```

```
      if (this.processQueue.length > 0) {
        this.totalExecutionTime += this.contextSwitchTime;
        await interfaceManager.indicateContextSwitch(this.contextSwitchTime);
      }
    }
    interfaceManager.updateVisualQueue(this.processQueue);
  }
}

/* Como é o algoritmo  mais simples, apenas executa os processos tirando os primeiros da fila,
solicita que o gerenciador de interface faça as representações visuais, executa cada
um dos processos esperando o tempo de execução e de troca de contexto */

export class SJF extends FCFS {
  async run(interfaceManager) {
    this.processQueue.sort((a, b) => a.executionTime - b.executionTime);
    await super.run(interfaceManager);
  }
}

// Se aproveita do código do FCFS apenas ordenando os processos antes de começar a executá-los

export class RR extends Scheduler {
  constructor(processes, contextSwitchTime, quantum) {
    super(processes, contextSwitchTime);
    this.quantum = quantum;
  }

  async run(interfaceManager) {
    while (this.processQueue.length > 0) {
      interfaceManager.updateVisualQueue(this.processQueue);
      const process = this.processQueue.shift();
      const executionTime = Math.min(this.quantum, process.remainingTime);

      this.totalExecutionTime += executionTime;
      await process.execute(executionTime, interfaceManager);

      if (this.processQueue.length > 0) {
        this.totalExecutionTime += this.contextSwitchTime;
        await interfaceManager.indicateContextSwitch(this.contextSwitchTime);
      }
      if (!process.isComplete())
        this.processQueue.push(process);
    }
    interfaceManager.updateVisualQueue(this.processQueue);
  }
}
```

```
58
59  /* Agora não roda mais os processos até sua completude, mas sim, no máximo,
60  até a duração especificada pelo quantum. Com isso, é necessário ver se o processo foi
61  finalizado antes de colocá-lo ao fim da fila novamente */
62
63  export class PriorityRR extends Scheduler {
64    constructor(processes, contextSwitchTime, quantum) {
65      super(processes, contextSwitchTime);
66      this.quantum = quantum;
67    }
68
69    async run(interfaceManager) {
70      const priorityGroups = this.groupByPriority();
71
72      const sortedPriorityGroups = Object.keys(priorityGroups)
73        .sort((a, b) => b - a)
74        .map(priority => priorityGroups[priority]);
75
76      for (const processes of sortedPriorityGroups) {
77        const rr = new RR(processes, this.contextSwitchTime, this.quantum);
78        await rr.run(interfaceManager);
79        this.totalExecutionTime += rr.totalExecutionTime
80        await interfaceManager.indicateContextSwitch(this.contextSwitchTime);
81        this.totalExecutionTime += this.contextSwitchTime;
82      }
83      this.totalExecutionTime -= this.contextSwitchTime;
84    }
85
86    groupByPriority() {
87      const groups = {};
88      this.processQueue.forEach((process) => {
89        if (!groups[process.priority]) {
90          groups[process.priority] = [];
91        }
92        groups[process.priority].push(process);
93      });
94      return groups;
95    }
96  }
97
98  // Como é um RR por prioridade, primeiramente separa-os por prioridade para depois aplicar o RR
99  // em cada um desses grupos. Note que pode parecer complexa essa abordagem ao invés de utilizar
100 // algo similar à ordenação do SJF, porém sem essa separação por grupo para cada RR, um processo não
101 // finalizado de alta prioridade iria ao fim da fila toda e não apenas de seu grupo
102
```

O process é menos complexo:

```
1
2  class Process {
3    constructor(name, executionTime, priority) {
4      this.name = name;
5      this.percentageComplete = 0;
6      this.executionTime = executionTime;
7      this.remainingTime = executionTime;
8      this.priority = priority;
9    }
10
11   async execute(time, interfaceManager) {
12     return new Promise((resolve) => {
13       this.remainingTime -= time;
14       this.percentageComplete =
15         ((this.executionTime - this.remainingTime) / this.executionTime) * 100;
16         interfaceManager.startProcessVisually(this)
17       setTimeout(() => {
18         interfaceManager.endProcessVisually(this);
19         resolve();
20       }, time * 1000);
21     });
22   }
23
24   isComplete() {
25     return this.remainingTime <= 0;
26   }
27 }
28
29 /* Uma classe preocupada apenas com o status de sua execução, repassando
30 ao gerenciador seu andamento para que o usuário possa acompanhar */
31
32 export default class ProcessFactory {
33   static #counter = 0;
34
35   static createProcessWithValues(executionTime, priority) {
36     this.#counter += 1;
37     const name = `P${this.#counter}`;
38     return new Process(name, executionTime, priority);
39   }
40
41   static createCopiedProcess(process) {
42     return new Process(process.name, process.executionTime, process.priority);
43   }
44 }
45
46 // Encapsula a criação dos processos para que haja um controle da quantidade de processos criados
47
```

Como o InterfaceManager é bem extenso, será passada as assinaturas dos métodos com os mais importantes tendo uma explicação mais detalhada:

```
1
2  export default class InterfaceManager {
3      constructor() {
4          this.processQueue = [];
5      }
6
7      init() // inicializa os eventListeners
8
9      addProcess(executionTime, priority)
10     // adiciona os processos à sua lista de processos para poder exibir ao usuário
11
12     getScheduler(algorithmType, processQueue)
13     // retorna o scheduler correto, inicializado com os parametros necessários
14
15     async handleSimulationStart() {
16         this.clearAlgorithmTable();
17         const auxQueue = this.deepCopy(this.processQueue);
18         const algorithms = ["FCFS", "SJF", "RR", "PriorityRR"];
19         for (let algorithm of algorithms) {
20             this.processQueue = this.deepCopy(auxQueue);
21             const scheduler = this.getScheduler(algorithm, this.processQueue);
22             this.updateVisualAlgorithmName(algorithm);
23             if (scheduler) {
24                 await scheduler.run(this);
25                 this.clearProcessTable();
26                 this.logAlgorithmTime(algorithm, scheduler.totalExecutionTime);
27             }
28         }
29         this.processQueue = []
30     }
31
32     /* responsável pelo inicio da simulação, rodando cada um dos algoritmos implementados
33     como JS utiliza copia por referência, utiliza um método próprio para que possa ser
34     ter a execução de uma fila com os mesmos processo sem que a finalização num algoritmo
35     anterior interfira */
36
37     deepCopy(original) // retorna uma array com uma cópia sem refeência
38
39     clearProcessTable()
40     // esvazia a tabela de processos para que se tenha os novos dados de forma limpa
41
42     logAlgorithmTime(algorithm, totalExecutionTime) // registra os tempos de cada algoritmo
43
44     updateVisualAlgorithmName(algorithmName) // mostra qual algortimo está sendo rodado
45
```

```
46      handleReset() // responsável por limpar os dados para se reiniciar a simulação
47
48      clearAlgorithmTable()
49      // esvazia a tabela de algoritmos para que se tenha os novos dados de forma limpa
50
51      updateSliderValue(slider, outputId) // altera o valor exibido ao usuário em tempo real
52
53      handleProcessAddition() // lida com o evento de adicionar processos
54
55      addSliderListener(sliderId, outputId) // cria o evento de ouvir a interação do usuário
56
57      addProcessToVisualQueue(process) // evidencia a criação do processo ao usuário
58
59      async indicateContextSwitch(contextSwitchTime)
60      // indica ao usuário que há uma troca de contexto para que saiba que está executando
61
62      startProcessVisually(process) // indica ao usuário que um processo começou a executar
63
64      endProcessVisually(process) // indica ao usuário que um processo terminou de executar
65
66      updateVisualQueue(processQueue) // altera a visualização da fila enquando um processo é executado
67  }
68
69
```

Dessa forma, toda interação do usuário é repassada ao InterfaceManager para que ele tenha uma resposta visual em tempo real e possa acompanhar a simulação, sendo as classes de process e scheduler focadas apenas em seu comportamento interno

4 Funcionamento do simulador

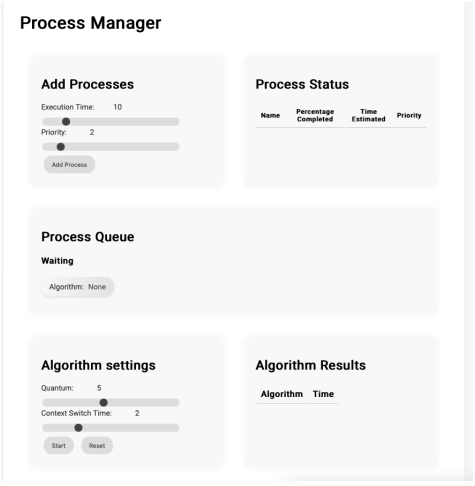

Figura 1: Inicio do simulador

Para utilizar, basta escolher as configurações dos algoritmos e adicionar quantos processos quiser, podendo alterar a duração de cada processo e sua prioridade, sendo ela representada visualmente para que se enxergue claramente a prioridade

de cada uma Durante a execução do simulador, é inserida na tabela de processos todos dados relevantes quanto a ele para

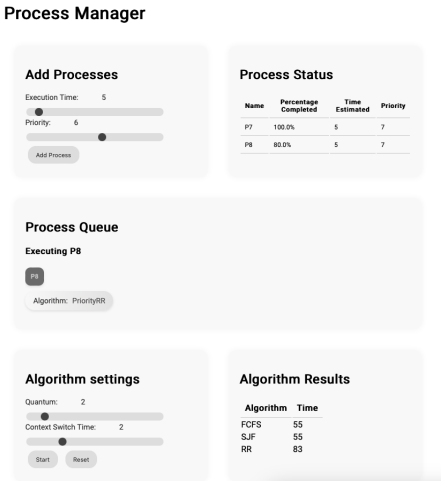

Figura 2: Execução do simulador

que se possa acompanhar a execução do processo, bem como os tempos de cada algoritmo

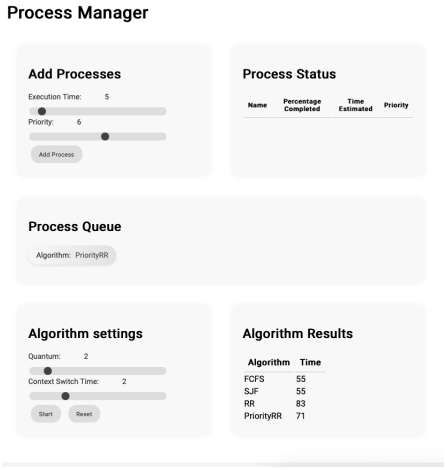

Figura 3: Fim da simulação

Note que também é possível simular pela página online do github

5 Conclusão

A aplicação web desenvolvida, que analisa, calcula e compara o tempo de execução de diferentes algoritmos de escalonamento, representa um passo importante na visualização e compreensão desses métodos. Permitindo aos usuários inserir processos com tempos de execução e prioridades, a ferramenta oferece uma análise direta e comparativa de como os algoritmos FCFS, RR, SJN e PriorityRR funcionam em diferentes cenários. A capacidade de identificar qual algoritmo demora mais em condições específicas fornece insights úteis que podem ser aplicados em diversos contextos. O projeto, portanto, não só facilita o entendimento desses algoritmos complexos, mas também serve como uma referência prática para estudantes, educadores e profissionais na área de ciência da computação.