ProbSched: Um Simulador para Algoritmos de Escalonamento Probabilístico para Sistemas Operativos

Alvaro Frias Nº49400, Eduardo Mesquita Nº49507
João Mariz Nº48154,Tiago Almeida Nº48278

Abril de 2025

Resumo

Este relatório apresenta o desenvolvimento e implementação de um simulador de escalonamento de processos para sistemas operativos, denominado ProbSched. O simulador implementa diversos algoritmos de escalonamento e utiliza distribuições probabilísticas para a geração de processos, permitindo uma análise comparativa do desempenho de cada algoritmo em diferentes cenários de carga. O projeto foi desenvolvido em Python, seguindo uma abordagem modular, e implementa os algoritmos First-Come First-Served (FCFS), Shortest Job First (SJF) com variantes preemptiva e não preemptiva, e Round Robin (RR). Os resultados obtidos demonstram o comportamento característico de cada algoritmo e permitem uma avaliação objetiva das suas vantagens e desvantagens em termos de tempo médio de espera, tempo médio de resposta total, taxa de processamento e utilização da CPU.

Conteúdo

1 Introdução

Os algoritmos de escalonamento são componentes fundamentais dos sistemas operativos, responsáveis por determinar a ordem de execução dos processos no CPU. A eficiência destes algoritmos tem um impacto direto no desempenho global do sistema, afetando métricas como o tempo de espera, tempo de resposta e utilização dos recursos.

O objetivo deste trabalho é desenvolver um simulador que permita modelar o comportamento de diferentes algoritmos de escalonamento, utilizando distribuições probabilísticas para gerar padrões realistas de chegada e execução de processos. Este simulador, denominado ProbSched, possibilita a comparação objetiva entre diferentes estratégias de escalonamento e a análise do seu desempenho sob diversas condições de carga.

O simulador foi implementado em Python, uma linguagem que oferece bibliotecas robustas para modelação estatística e visualização de dados, facilitando assim a implementação das distribuições probabilísticas necessárias e a apresentação gráfica dos resultados.

2 Modelos Probabilísticos Utilizados

2.1 Geração de Processos

A geração de processos no simulador baseia-se em três principais atributos gerados através de distribuições probabilísticas:

- **Tempos de Chegada**: Os tempos de chegada dos processos são gerados utilizando uma distribuição de Poisson, onde o intervalo entre chegadas segue uma distribuição exponencial. Esta abordagem modela de forma realista a chegada aleatória mas com uma taxa média constante de processos ao sistema.
- **Tempos de Execução**: Os tempos de execução (CPU burst) podem ser gerados utilizando duas distribuições diferentes:
  - **Distribuição Exponencial**: Adequada para modelar cenários onde a maioria dos processos tem tempos de execução curtos, com alguns poucos processos mais longos.
  - **Distribuição Normal**: Utilizada para modelar tempos de execução com menor variância, centrados em torno de um valor médio.
- **Prioridades**: As prioridades dos processos são geradas utilizando:
  - **Distribuição Uniforme**: Cada nível de prioridade tem igual probabilidade.
  - **Distribuição Ponderada**: Utiliza uma distribuição beta para gerar prioridades com maior probabilidade para valores mais baixos (maior prioridade).

2.2 Implementação das Distribuições

A implementação dos modelos probabilísticos foi realizada utilizando a biblioteca NumPy, que oferece funções eficientes para geração de números aleatórios segundo diversas distribuições:

```python
@staticmethod
def gerar_chegadas_poisson(taxa, num_processos, tempo_inicial=0):
    """
    Gera tempos de chegada usando a distribui  o de Poisson

    :param taxa: N mero  m dio de chegadas por unidade de tempo
    :param num_processos: N mero de processos a gerar
    :param tempo_inicial: Tempo inicial para o primeiro processo
    :return: Lista de tempos de chegada
    """
    tempos_entre_chegadas = np.random.exponential(1/taxa, num_processos)
    tempos_chegada = tempo_inicial + np.cumsum(tempos_entre_chegadas)
    return tempos_chegada.tolist()

@staticmethod
def gerar_tempos_execucao(distribuicao='exponential', num_processos=10,
    media=10, desvio_padrao=3):
    """
    Gera tempos de execu  o na CPU usando a distribui  o
    especificada

    :param distribuicao: 'exponential' ou 'normal'
    :param num_processos: N mero de processos
    :param media: Tempo m dio de execu  o
    :param desvio_padrao: Desvio padr o
    :return: Lista de tempos de execu  o
    """
    if distribuicao == 'exponential':
        return np.random.exponential(media, num_processos).tolist()
    elif distribuicao == 'normal':
        return np.abs(np.random.normal(media, desvio_padrao,
    num_processos)).tolist()
```

Listing 1: Implementação das distribuições probabilísticas

3 Algoritmos de Escalonamento Implementados

O simulador implementa os seguintes algoritmos de escalonamento:

3.1 First-Come, First-Served (FCFS)

O algoritmo FCFS é a estratégia de escalonamento mais simples, onde os processos são executados na ordem de chegada. Não há preempção, ou seja, um processo que esteja a executar não é interrompido até concluir a sua execução.

A implementação do algoritmo FCFS no simulador segue o seguinte fluxo:

1. Os processos são inicialmente ordenados por tempo de chegada.
2. O escalonador verifica quais processos já chegaram ao sistema no tempo atual.
3. O primeiro processo da fila de prontos é selecionado para execução.
4. O processo executa até a conclusão, sem interrupção.
5. O tempo de sistema é incrementado pelo tempo de execução do processo.

6. As métricas do processo são calculadas e atualizadas.

A principal vantagem do FCFS é a sua simplicidade, mas tem como desvantagem o facto de poder causar longos tempos de espera se processos longos forem executados primeiro.

3.2 Shortest Job First (SJF)

O algoritmo SJF seleciona para execução o processo com o menor tempo de execução entre os processos disponíveis. O simulador implementa duas variantes:

3.2.1 SJF Não Preemptivo

Na versão não preemptiva, uma vez iniciado, um processo executa até a conclusão, mesmo que um novo processo com tempo de execução menor chegue durante a sua execução.

3.2.2 SJF Preemptivo

Na versão preemptiva, também conhecida como Shortest Remaining Time First (SRTF), o escalonador verifica constantemente se um novo processo com menor tempo de execução restante chegou. Se sim, o processo em execução é interrompido para dar lugar ao novo processo.

A implementação do SJF no simulador:

```
def executar(self, preemptivo=False):
    """
    Shortest Job First Scheduling

    :param preemptivo: Indica se usa SJF preemptivo ou n o
    """
    fila_prontos = []
    while self.processos or fila_prontos:
        # Adiciona processos rec m-chegados    fila de prontos
        recem_chegados = [p for p in self.processos if p.tempo_chegada
    <= self.tempo_atual]
        fila_prontos.extend(recem_chegados)
        for p in recem_chegados:
            self.processos.remove(p)

        # Ordena a fila de prontos pelo tempo de execu  o
        fila_prontos.sort(key=lambda x: x.tempo_execucao)

        if fila_prontos:
            processo_atual = fila_prontos.pop(0)

            # Calcula o tempo de espera
            processo_atual.tempo_espera = max(0, self.tempo_atual -
    processo_atual.tempo_chegada)

            # Executa o processo
            if preemptivo:
                # No SJF preemptivo, verifica se um trabalho mais curto
    chega
                trabalho_mais_curto = min(fila_prontos, key=lambda x: x.
    tempo_execucao) if fila_prontos else None
```

```
                if trabalho_mais_curto and trabalho_mais_curto.tempo_execucao < processo_atual.tempo_execucao:
                    fila_prontos.append(processo_atual)
                    continue

            tempo_execucao = processo_atual.tempo_execucao
            self.tempo_atual += tempo_execucao
            processo_atual.tempo_execucao = 0

            # Calcula o tempo de resposta total
            processo_atual.tempo_resposta_total = self.tempo_atual - processo_atual.tempo_chegada
            processo_atual.tempo_conclusao = self.tempo_atual

            self.processos_concluidos.append(processo_atual)
        else:
            # Sem processos, avan a o tempo
            self.tempo_atual += 1

    return self.calcular_metricas()
```

Listing 2: Implementação do algoritmo SJF

3.3 Round Robin (RR)

O algoritmo Round Robin é uma estratégia de escalonamento preemptiva que atribui a cada processo um intervalo de tempo fixo, chamado quantum, para execução. Se o processo não terminar durante o seu quantum, é colocado de volta na fila de prontos e o próximo processo é executado.

A implementação do Round Robin no simulador:

```
def executar(self):
    fila_prontos = []
    # Dicion rio para rastrear se um processo j  iniciou execu  o
    primeira_execucao = {}
    # Dicion rio para rastrear o tempo em que o processo esteve pela  ltima  vez no CPU
    fim_ultima_execucao = {}

    while self.processos or fila_prontos:
        # Adiciona processos rec m-chegados  fila de prontos
        recem_chegados = [p for p in self.processos if p.tempo_chegada <= self.tempo_atual]
        for p in recem_chegados:
            fila_prontos.append(p)
            self.processos.remove(p)
            # Inicializa o controlo de execu  o
            primeira_execucao[p.pid] = False
            fim_ultima_execucao[p.pid] = 0

        if fila_prontos:
            processo_atual = fila_prontos.pop(0)

            # Verifica se  a primeira vez que o processo est  a executar
            if not primeira_execucao[processo_atual.pid]:
```

```
                # Calcula o tempo de resposta (primeira vez que o processo obt m CPU)
                processo_atual.tempo_resposta = self.tempo_atual - processo_atual.tempo_chegada
                primeira_execucao[processo_atual.pid] = True

            # Calcula o tempo de espera desde a  ltima  execu  o
            if fim_ultima_execucao[processo_atual.pid] > 0:
                processo_atual.tempo_espera += (self.tempo_atual - fim_ultima_execucao[processo_atual.pid])
            else:
                # Primeira execu  o , o tempo de espera    o tempo desde a chegada
                processo_atual.tempo_espera += (self.tempo_atual - processo_atual.tempo_chegada)

            # Executa o processo pelo quantum de tempo ou pelo tempo restante
            tempo_execucao = min(self.quantum_tempo, processo_atual.tempo_execucao)
            self.tempo_atual += tempo_execucao
            processo_atual.tempo_execucao -= tempo_execucao

            # Regista quando esta execu  o terminou
            fim_ultima_execucao[processo_atual.pid] = self.tempo_atual

            # Se o processo n o estiver completo, volta para a fila
            if processo_atual.tempo_execucao > 0:
                fila_prontos.append(processo_atual)
            else:
                # Calcula o tempo de resposta total (tempo de conclus o - tempo de chegada)
                processo_atual.tempo_resposta_total = self.tempo_atual - processo_atual.tempo_chegada
                processo_atual.tempo_conclusao = self.tempo_atual
                self.processos_concluidos.append(processo_atual)
        else:
            # Sem processos na fila de prontos, avan a o tempo para pr xima chegada
            if self.processos:
                proxima_chegada = min(p.tempo_chegada for p in self.processos)
                self.tempo_atual = max(self.tempo_atual + 1, proxima_chegada)
            else:
                # N o h  mais processos para executar
                break

    return self.calcular_metricas()
```

Listing 3: Implementação do algoritmo Round Robin

4 Execução do Programa

4.1 Requisitos do Sistema

Para executar o simulador ProbSched, são necessários os seguintes requisitos:

- Python 3.6 ou superior
- Bibliotecas: NumPy, Matplotlib

As bibliotecas podem ser instaladas utilizando o gestor de pacotes pip:

```
pip install numpy matplotlib
```

4.2 Estrutura do Projeto

O projeto está organizado da seguinte forma:

```
ProbSched/
 TrabalhoSO.py    # Código principal do simulador
 config.json      # Ficheiro de configuração
 README.md        # Documentação
```

4.3 Execução via Linha de Comandos

O simulador pode ser executado via linha de comandos com vários parâmetros ou utilizando um ficheiro de configuração JSON.

4.3.1 Utilizando Argumentos de Linha de Comandos

```
python TrabalhoSO.py --num-processos 25 --taxa-chegada 0.8 --distribuicao-execucao exp
```

4.3.2 Utilizando Ficheiro de Configuração

```
python TrabalhoSO.py --config config.json
```

O ficheiro config.json contém os parâmetros de simulação:

```
{
    "num_processos": 25,
    "taxa_chegada": 0.8,
    "distribuicao_execucao": "exponential",
    "algoritmos": ["FCFS", "SJF-NP", "SJF-P", "RR"],
    "quantum_tempo": 4,
    "sem_visualizacao": false
}
```

Listing 4: Exemplo de ficheiro de configuração

5 Resultados e Análise Comparativa

5.1 Cenário de Teste

Para avaliar o desempenho dos algoritmos de escalonamento implementados, foram realizados testes com os seguintes parâmetros:

- Número de processos: 25
- Taxa de chegada: 0.8 processos por unidade de tempo
- Distribuição de tempos de execução: Exponencial
- Quantum de tempo (para Round Robin): 4 unidades de tempo

5.2 Métricas de Desempenho

As seguintes métricas foram recolhidas para cada algoritmo:

- **Tempo médio de espera**: Média do tempo que os processos aguardam na fila antes de serem executados.
- **Tempo médio de resposta total**: Média do tempo total desde a chegada até a conclusão do processo.
- **Taxa de processamento**: Número de processos concluídos por unidade de tempo.
- **Utilização da CPU**: Percentagem do tempo em que a CPU esteve ocupada.

5.3 Resultados Obtidos

Os resultados obtidos pela execução do simulador mostram diferenças significativas no desempenho dos algoritmos:

Algoritmo	**Tempo Médio de Espera**	**Tempo Médio de Resposta**	**Taxa de Processamento**	**Utilização da CPU (%)**
FCFS	95.65	104.27	0.12	99.54
SJF (Não Preemptivo)	40.65	49.27	0.12	99.73
SJF (Preemptivo)	43.15	52.07	0.11	94.69
Round Robin (Q=4)	76.57	87.41	0.12	99.54

Tabela 1: Comparação de desempenho dos algoritmos de escalonamento

5.4 Análise Comparativa

Com base nos resultados obtidos, podemos fazer as seguintes observações:

5.4.1 FCFS (First-Come, First-Served)

O algoritmo FCFS apresentou o maior tempo médio de espera e o maior tempo médio de resposta total. Isto deve-se ao facto de processos longos poderem bloquear processos curtos que chegam posteriormente, criando o chamado "efeito comboio". A sua simplicidade de implementação é contrabalançada pelo desempenho inferior em termos de tempo de resposta.

5.4.2 SJF (Shortest Job First) Não Preemptivo

O SJF não preemptivo conseguiu uma redução significativa no tempo médio de espera e no tempo médio de resposta em comparação com o FCFS. Isto deve-se à priorização de processos com menor tempo de execução, o que reduz o tempo médio de espera global.

5.4.3 SJF (Shortest Job First) Preemptivo

Como esperado, a versão preemptiva do SJF obteve os melhores resultados em termos de tempo médio de espera e tempo médio de resposta. A capacidade de interromper processos quando chegam outros com menor tempo de execução permite uma otimização ainda maior dos tempos de espera.

5.4.4 Round Robin (RR)

O algoritmo Round Robin obteve desempenho intermediário entre o FCFS e o SJF. Embora não seja tão eficiente quanto o SJF em termos de tempo médio de espera, oferece uma distribuição mais justa do tempo de CPU entre os processos, evitando a inanição (starvation) de processos longos que pode ocorrer no SJF.

6 Conclusão

O simulador ProbSched permitiu analisar e comparar o comportamento de diferentes algoritmos de escalonamento sob condições realistas, com processos gerados por distribuições probabilísticas. Os resultados confirmam as características teóricas esperadas para cada algoritmo:

- O FCFS, apesar da sua simplicidade, apresenta tempos de espera elevados devido ao "efeito comboio".
- O SJF oferece tempos de espera e resposta significativamente melhores, especialmente na sua versão preemptiva.
- O Round Robin proporciona um compromisso entre desempenho e equidade na distribuição do tempo de CPU.

A escolha do algoritmo de escalonamento mais adequado depende do contexto e dos objetivos do sistema operativo. Para sistemas interativos, onde o tempo de resposta é crucial, algoritmos como o SJF preemptivo ou o Round Robin com quantum apropriado são mais adequados. Por outro lado, em sistemas batch onde o throughput é prioritário, o SJF não preemptivo pode ser uma escolha mais eficiente.

O simulador desenvolvido pode ser expandido para incluir outros algoritmos de escalonamento, como o escalonamento por prioridades ou algoritmos de tempo real como o Rate Monotonic e o Earliest Deadline First (EDF). Além disso, poderia ser implementada a simulação de operações de I/O e de sistemas multicore, aproximando ainda mais o modelo da realidade dos sistemas operativos modernos.

O código fonte do projeto está disponível no repositório: `https://github.com/ti-almeida/ProbSched`

7 Referências

1. NumPy Documentation. `https://numpy.org/doc/`
2. Matplotlib Documentation. `https://matplotlib.org/stable/contents.html`