Trabalho Prático da disciplina de Estrutura de Dados

Leticia Ribeiro Miranda
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
leticia-ribeiro98@hotmail.com

1. Introdução

O objetivo desse anexo é desenvolver uma otimização para o sistema de filas de um determinado hospital.

A análise de complexidade de tempo feita anteriormente, possibilitou a identificação de dois métodos que são chamados repetidamente na simulação e que são os principais responsáveis pelo desempenho do programa, são eles: '*insereEvento()'* e *'retiraProximoEvento'*. Com base nessa observação, a estratégia principal para otimizar o desempenho foi reduzir a frequência com que esses métodos são invocados. Para alcançar isso, os pacientes que recebem alta logo após o atendimento ou que não necessitam de determinados procedimentos deixam de ser inseridos no escalonador, evitando a chamada desnecessária dos dois métodos e, consequentemente, otimizando o desempenho do sistema.

2. Implementação

Para a implementação da melhoria, foram inseridas três verificações ao longo do programa. A primeira delas é logo após o atendimento, caso o paciente tenha alta, ele imediatamente vai para o caso 7 e não é mais inserido ou removido dos eventos.

As outras duas estão implementadas nas funções *'atualizarEstadoPaciente'* e *'liberarProcedimentoEProcessarProximo'*. A duração do procedimento para um determinado paciente é calculada ao multiplicar a quantidade de vezes que ele realiza o procedimento pelo tempo médio de execução correspondente. Por exemplo, o paciente 008 passa por medidas hospitalares duas vezes, com um tempo médio de execução de 0,5 horas. Assim, a duração total desse procedimento para ele será 2 * 0,5 = 1 hora.

Se a quantidade de vezes que o paciente realiza um determinado procedimento for igual a zero, a duração total desse procedimento será igualmente zero. Nesse caso, o paciente avança automaticamente para o próximo procedimento. Esse processo de verificação se repete até que ele receba alta, otimizando o sistema ao reduzir a inserção de eventos. Portanto, um evento só é inserido caso o paciente realmente precise realizar o procedimento.

Estas soluções propostas evitam a inserção e remoção de eventos desnecessários.

2.1. Modificação do código

No código, substitua a função na linha 70 pela função da linha 434 e a função na linha 107 pela da linha 445. Além disso, descomente a condição que vai da linha 309 até a linha 314.

3. Análise Experimental

Na etapa de análise experimental foram utilizadas três bibliotecas para medir o tempo de relógio e recursos do programa.

Para medir o tempo de relógio, foi usada a função ‘high_resolution_clock::now()’ da biblioteca chrono, que registra o tempo inicial e o tempo final ao redor da execução da função alvo. A diferença entre esses dois instantes é calculada e armazenada em uma variável do tipo ‘duration<double, std::milli>’, que converte a duração para milissegundos.

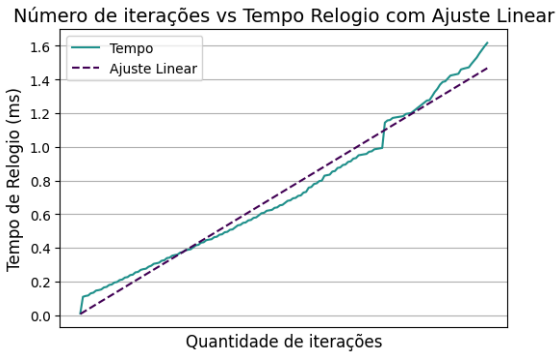
Já para os tempos de utilização de recursos (usuário e sistema), usou-se a função clock_gettime que pertence à biblioteca padrão do sistema no Linux, ‘time.h’. Calculou-se a diferença entre o início e fim da execução da função alvo para obter as durações.

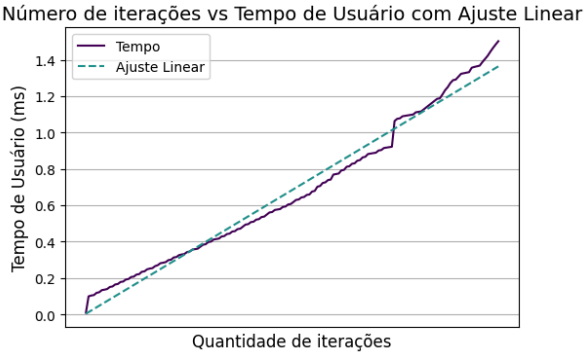
Além disso, foram utilizadas as ferramentas Cachegrind, que simula a hierarquia de memória e mede o desempenho de cache, e o Valgrind, que analisa as funções mais chamadas.

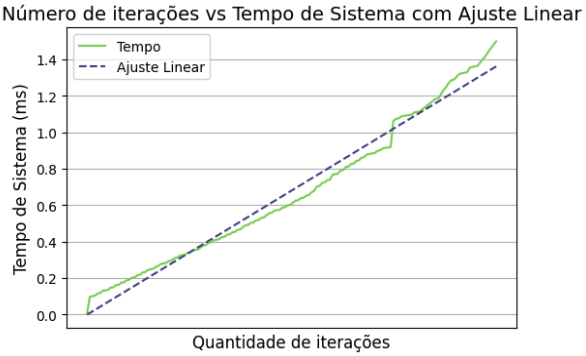
Por fim, em todas as análises, utilizaram-se uma base de dados com 1000 registros, totalizando 36.758 mil caracteres. Após esse limite, o programa apresentou um erro de segmentation fault, o que restringiu a quantidade de dados analisados.

Assim como no código anterior, observa-se uma tendência linear entre a quantidade de iterações e o tempo que não reflete a complexidade de tempo do programa, isso ocorre pela quantidade pequena de dados.

No entanto, ao comparar os tempos gerados antes e depois das modificações, observando o eixo y, é possível perceber uma redução significativa nesses valores, refletindo uma melhora de desempenho.







Fazendo a regressão linear, pode-se concluir que, para cada incremento de 1 unidade no eixo x, o tempo aumenta, em média, 0.0094 milissegundos (ms) para de relógio e 0088 milissegundos (ms) para o de usuário e de sistema.

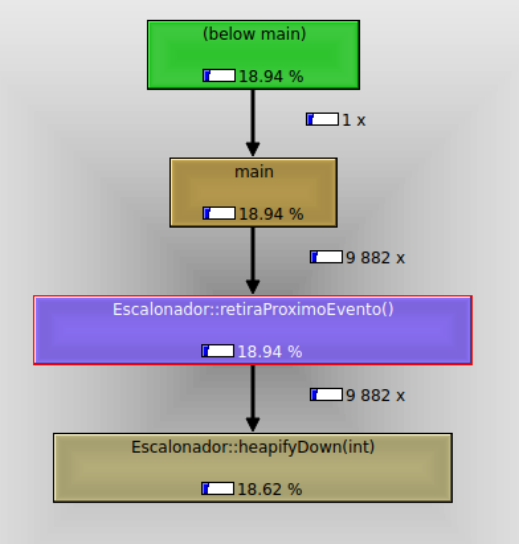
Executando o código utilizando o Cachegrind, especificamente com a ferramenta cg_annotate, foram analisadas as principais funções que impactam no total de instruções executadas.

Embora o programa tenha executado um total de 164.954.452 instruções, o que representa um aumento de 5% em relação ao algoritmo anterior, houve uma diminuição significativa nas instruções dedicadas ao escalonador, totalizando 40.415.159 (24,5%), o que representa uma redução de 35,63%, se comparado com o algoritmo anterior. Isso indica que a solução proposta foi eficaz.

Dentro desta classe, o método *'heapifyDown(int)'* ainda continua sendo o maior responsável pelo consumo de instruções, representando 17,5% (28.824.794) do total desse valor. Já o *'heapifyUp(int)'* vem em segundo lugar, consumindo 4,6% (7.513.114) de instruções. Ambos apresentaram uma redução significativa de 30,81% e 50,71%, respectivamente, quando comparados ao programa anterior.

Por fim, através do comando *'kcachegrind callgrind.out.<pid>'*, analisou-se as funções mais chamadas e quanto tempo de CPU foi consumido por cada uma delas.

Na figura a seguir, observa-se que o método *'retiraProximoEvento()'* apresentou uma redução de 34,19% no tempo total de CPU. Na melhoria proposta, ele representa 18,94% do tempo total de execução. Por outro lado, o tempo de CPU dedicado à função *'insereEvento()'* foi tão reduzido que não apareceu nas visualizações do KCachegrind



Esse resultado demonstra que as melhorias implementadas tiveram um impacto significativo no escalonador. Como este é o principal fator responsável pela complexidade de tempo do programa, a redução no tempo de CPU desse método torna o processo mais eficiente,

indicando que o programa, como um todo, se tornou mais rápido e capaz de executar suas tarefas com maior desempenho.

4. Conclusão

Com base nos resultados apresentados, é possível concluir que as melhorias implementadas no sistema de escalonamento foram bem-sucedidas em otimizar o desempenho do programa. As alterações realizadas na redução de chamadas desnecessárias dos métodos *'insereEvento()'* e *'retiraProximoEvento()'*, resultaram em uma diminuição significativa no consumo de instruções e no tempo total de CPU dedicado ao escalonador. Especificamente, o método *'retiraProximoEvento()'* teve uma redução de 34,19% no tempo de CPU, enquanto o impacto do método *'insereEvento()'* foi minimizado a ponto de não aparecer nas visualizações do KCachegrind.

Ademais, a análise experimental demonstrou que, embora o número total de instruções executadas pelo programa tenha aumentado em 5%, houve uma redução expressiva nas instruções dedicadas ao escalonador, evidenciando que a solução foi eficaz em focar nos elementos mais críticos para o desempenho do sistema.

Portanto, as otimizações implementadas tornaram o escalonador mais eficiente e contribuíram para a melhoria geral do desempenho do programa, tornando-o mais rápido e eficaz no gerenciamento do sistema de filas. Os resultados obtidos validam a abordagem adotada e destacam a importância de priorizar os componentes responsáveis pela maior parte da complexidade de tempo em sistemas críticos como este.

1. Bibliografia

Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.