SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho 1 - AEDS 1

Sistema de gerenciamento de processos utilizando lista de cursores

INGRED FONSECA DE ALMEIDA [EF 04691]
MARIA CLARA BRAGA ARAÚJO VIANA [EF04667]
RAFAELA TOLEDO DOLABELLA [EF04665]

Florestal - MG
2021

Sumário

1. Introdução

Neste trabalho foram implementados dois Tipos Abstratos de Dados (TADs). Um para criação do processo, com Identificador do Processo (PID), horário de criação e prioridade, para que um sistema operacional faça gerência de processos ativos. Utilizando o segundo TAD para fazer uma lista, no formato de vetor de processos, duplamente encadeada através de cursores [1]. Desse modo, com a integração dos TADs, o código criado deverá chamar as funções principais, que são a cria lista, insere e remove itens, e com isso, pela main, pode-se escolher se a entrada de dados será por entrada padrão ou por arquivos.

Pela entrada padrão, como saída no terminal, deverá ser printado a lista de PIDs com sua prioridade, e ao final será mostrado o tempo de execução. Agora, caso a opção escolhida seja a entrada por arquivo, a saída será um novo arquivo que conterá o número do teste e também o tempo de execução de inserção e remoção dos processos na lista.

2. Organização

Na Figura 1 é possível visualizar a organização do projeto. Na pasta nomeada como pedido no comando do Trabalho Prático (TP), com o nome e matrícula das integrantes e o número do TP, temos:

- O código do programa, constituído por um arquivo main.c, o TAD_Processo (.h e .c) e o TAD_Lista (também .h e .c), com todas as funções e estruturas de dados solicitadas;
- Os arquivos de teste, enviados para os alunos, de 100.000, 200.000, 300.000, 400.000, 500.000 e 600.000 processos;

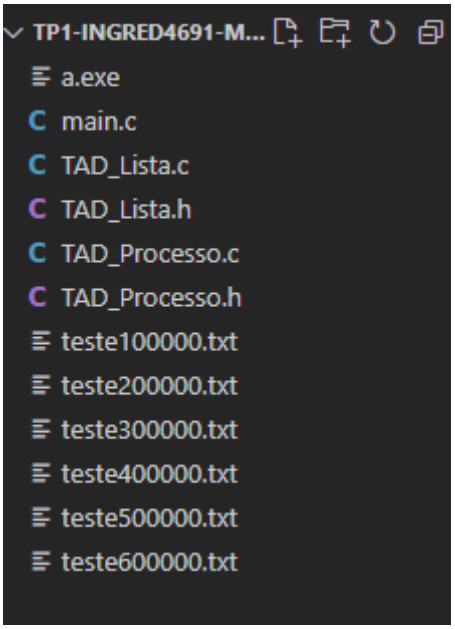

Figura 1 - Arquivos de código e de testes.

Junto a isso em uma pasta compactada há um arquivo PDF dessa documentação.

Para execução do programa é necessário compilar os arquivos .c, criando o arquivo executável (na imagem “a.exe”) que deve ser executado.

3. Desenvolvimento

3.1 TAD_Processo

Começamos com a criação do “*TAD_Processo”*, onde criamos uma estrutura do tipo Processo com dois itens do tipo inteiro: “*PID”* e “*prioridade”*. No comando do trabalho também era solicitado a hora atual, mas apesar de diversas tentativas não tivemos êxito com essa variável, como na atuação do restante do programa a falta dessa variável não interferia no funcionamento do código optamos por seguir em frente na prática.

- **FUNÇÕES**

Para cada um desses itens foi feito funções de leitura (*Get*) e alteração (*Set*). Ambos itens foram criados de forma aleatória. Além disso, durante a criação do "*PID"* fizemos um *if* para que não tivesse Identificadores de Processo iguais.

3.2 TAD_Lista

Primeiramente inserimos o "*TAD_Processo.h"* como biblioteca. Fizemos uma estrutura do tipo *"Celula"* com o item do tipo "*Processo*" e dois inteiros, "*prox*" e "*ante",* que guardariam o índice da célula próxima e anterior, respectivamente, para encadear nossa lista duplamente.

Em seguida criamos outra estrutura, essa do tipo "*Lista*", contendo, variáveis inteiras: O índice da primeira ("*primeiro*"), da última ("*ultimo*") célula e da primeira célula livre ("*primeira_celula_livre"*); a quantidade de células livres ("*qtd_celulas_livres"*); e um vetor do tipo "*Celula"* ("*vetor*").

- **FUNÇÕES**

void ***incializa_lista***:

Para iniciarmos a lista enviamos como parâmetro uma *"lista*" do tipo "*Lista"* e um valor inteiro "*N".* Alocamos dinamicamente a lista e o vetor de células através da função *malloc*, utilizando o *sizeof* e o valor "*N"* para separarmos o local no *Hea*p adequado. Como criamos a lista vazia, as variáveis "*primeiro*" e "*ultimo"* guardam o valor (-1), pois não é o índice de nenhuma célula já que o vetor se inicia no índice 0. Se nenhuma célula está ocupada temos nossa variável que indica a quantidade de células disponíveis igual ao valor de células do vetor ("*N"*) e a primeira célula livre é a primeira do vetor (a de índice 0). Para encadearmos as células livres chamaremos uma outra função *celulas_livres* e enviaremos como parâmetro o vetor de células e o inteiro "*N".*

void ***celulas_livres***:

Para encadear as células disponíveis e terminarmos a inicialização da nossa lista usaremos a função *for* para percorrer todo o vetor alocando uma célula nova em cada posição e aplicando para cada posição (i) (i-1) na posição "*ante"* e (i+1) na "*prox*", até chegarmos na última posição do vetor (N-1) e colocaremos como "*prox"* o valor (-1) pois depois dela não há outra célula disponível.

void ***insere_lista***:

Para inserirmos uma célula na lista precisamos que haja uma lista e que tenha células disponíveis nessa lista. Então para iniciarmos essa função fazemos essas duas verificações. Se não houver uma lista ainda chamamos a função *incialista_lista*. Já se não houver célula livre, printamos o erro para o usuário e encerramos a função.

Caso seja possível fazer a inserção criamos uma variável inteira que guardará o índice da primeira célula livre, preenchemos o item do vetor nessa posição e passamos o valor do "*prox"* da celula que acabamos de selecionar para a variável que indica o índice da primeira célula disponível. Além disso, criamos um inteiro para representar a quantidade de células ocupadas, que usaremos mais tarde, subtraindo a quantidade de células livres do valor total do vetor.

Teremos 6 situações de inserção e faremos uma codificação para cada uma delas. Em todas elas diminuiremos a quantidade de células livres, pois estaremos ocupando uma.

01) Quando a lista está vazia: A variável que indica o índice do primeiro item da lista e a que indica o índice do último deve guardar o índice "j", pois sendo o único ele será o primeiro e o último. Como não há nada antes do primeiro e nada depois do último, os campos "*ante"* e "*prox*" do vetor na posição "j" devem guardar (-1).

Optamos por começar a comparação para a ordenação pelo final. Para isso, guardamos o índice da última célula ocupada com "*x*". Também guardamos o índice do primeiro em um inteiro "*z*" (essa variável será necessária em um dos casos que serão mostrados ainda).

Fizemos um *do while* para percorrermos todo o vetor, pois somente na primeira posição o "*ante*" é (-1), ou se entrar em um dos *if* que ao final terão um *break*. Dentro do *do while* primeiro colocamos a célula do vetor na posição "*x*" na célula "*aux*".

Comparamos o "*PID"* da célula na posição *"x"* com o da posição "*j*". Se a primeira for maior o *"x"* assume o valor do *"ante"* de *"x"* para que quando voltar para o início da estrutura de repetição o "*PID*" da nova célula seja comparado com o valor de "*PID*" menor mais próximo seguindo o encadeamento. Se com essa alteração, *"x"*

for (-1) isso quer dizer que o item da posição "*j*" é menor que o do primeiro item, assim teremos nosso segundo caso:

02) Quando o "*PID*" da nova célula é o menor "*PID*": Nesse caso a nova célula será a primeira. Então passaremos o "apontador" do primeiro para o vetor na posição "*j*". Como primeiro, seu *"ante"* deve ser (-1) e o seu "*prox"* será o antigo primeiro, no qual tínhamos separado o índice na variável *"z"*. Além disso, o *"ante"* de *"z"* muda de (-1) para o índice de "*j*".

Agora, caso o "*PID"* da célula na posição *"x"* for menor que o da posição "*j*" temos mais 3 opções de ações.

Criamos a variável "*ultimo"* que terá o mesmo valor do índice indicado pela variável "*ultimo"* da estrutura lista.

04) Caso o "*PID"* da célula na posição "*j*" for maior que o "*ultimo"*: O inteiro *"ultimo"* da da lista assumira o valor "*j*"; a célula da posição "*j*" terá seu *"ante"* como o "*ultimo"* (variável recém criada), seu "*prox"* terá o valor (-1), e o "*prox"* da célula na posição "*ultimo"* assumirá o valor de "*j*".

Criamos dois inteiros "*aux_prox"* e "*aux_ante*". Em seguida o "*aux_prox*" receberá o valor do "*prox"* da célula na posição *"x"*.

05) Caso o "*aux_prox*" for igual a -1: O *"prox"* da posição *"x"* e o *"ante*" da posição *"aux_prox"* receberam *"j"*. E a célula na posição *"j"* terá seu "*prox"* igual ao "*aux_prox*" e seu "ante" igual a *"aux_ante*".

06) Caso "*aux_prox"* for diferente de (-1): O "*aux_ante*" receberá o anterior do "*aux_prox"*, pois o "*j*" ficará entre esses dois valores. Por isso o "*prox"* da célula no índice *"x"* e o *"ante"* da célula na posição "*aux_prox"* receberam "*j*", e o "*prox"* de "*j*" receberá "*aux_prox"* e seu *"ante"* receberá "*aux_ante*".

void ***retira_primeiro_item***

Para removermos o primeiro item da lista, é necessário primeiramente fazer algumas verificações. Por exemplo, precisamos verificar se a lista tem apenas um item inserido, se sim, temos que alterar as variáveis ***primeiro*** e ***último***, de forma que apontem agora para -1, pois não haverá nenhum elemento inserido na lista. Também devemos nos lembrar de armazenar o ***anterior*** e o ***próximo*** do item que

estamos removendo, para não perdermos o acesso a essa célula, e criar variáveis auxiliares que irão armazenar esses valores de forma a estarem disponíveis quando ligarmos a célula retirada ao vetor de células livres. Outro caso que deve ser levado em consideração, é o fato de não haver mais células livres, se isso ocorrer, a célula que foi retirada deve ter seus cursores apontando para -1 pois não está ligada a mais nada. De modo contrário, se ainda houver células livres a quem se ligar, a célula retirada se torna a primeira célula livre e a antiga primeira célula livre agora tem seu anterior apontando para a nova primeira célula.

void ***imprime_vetor***

Para imprimirmos o vetor, é necessário verificarmos se o vetor está vazio, se sim, a verificação executa um return. Se não, fazemos a inicialização do vetor e em seguida armazenamos em uma variável contador, a variável ***primeiro***, pois é necessário percorrermos o vetor, e faremos isso da primeira posição até a última (por isso a necessidade de se armazenar o valor de ***primeiro***). Para realizar essa interação, utilizamos a estrutura ***do while,*** e o laço se executará enquanto o ***próximo*** de cada ***vetor[i]->prox*** for diferente de -1. Dessa forma, dentro deste loop, temos um printf que imprime na tela o valor do PID e prioridade de cada célula do vetor. Porém, é necessário estar atento para o fato de que a última célula tem seu ***próximo*** com valor -1, dessa forma a última célula não entra no loop criado, para resolvermos, é necessário implementarmos um printf, igual ao que está dentro do for, para esta última célula, mas com o vetor[lista->último]->item->prioridade para a prioridade, e vetor[i]->item->PID para o PID.

3.3 Main

- **ENTRADA PADRÃO**

  A escolha de entrada de dados é feita através de um scanf onde definimos que 0 seria entrada padrão e 1 seria entrada por arquivo. Em seguida, implementamos a estrutura switch case, onde case 0 é padrão e case 1 é arquivo. Caso o usuário tenha digitado 0, será solicitado em seguida que ele digite o valor da variável N, que é o tamanho do vetor. Em seguida, que ele digite o valor da variável vezes_insere e posteriormente, da variável vezes_retira, para que o programa chame as funções supracitadas quantas

vezes o usuário quiser. Após as entradas dos valores necessários, o arquivo chama a função inicializa_lista e a executa, passando como parâmetro a própria lista, criada com o tipo Lista *lista, e o N, que é o tamanho do vetor. Continuando, fizemos um for que chama a função insere_item e cria_processo, que vai de i=0 até i<vezes_insere, dessa forma criando um novo item e inserindo-o logo após a cada volta do laço, e o mesmo ocorre para a função remove_primeiro_item. E por fim, é chamada a função imprime_vetor, tendo a lista como parâmetro também. Toda essa estrutura, está contida entre uma variável beggin, no começo do programa, e uma variável end, que pega respectivamente o tempo incial e o tempo final, e calcula qual foi o tempo de execução do código, tudo isso através das funcionalidades da biblioteca ***time.h.*** Para encerrar, o programa imprime o valor do tempo de execução e a execução é encerrada.

- **ENTRADA POR ARQUIVO [2]**

  Caso o usuário tenha digitado 1 ele receberá uma mensagem no prompt de comando pedindo o nome do arquivo que deve ser aberto. Esse valor será guardado no vetor do tipo char "*teste_txt*". Abrimos o arquivo na variável "*teste"* do tipo FILE através da função *fopen* para leitura, já que seu segundo parâmetro é "*r*" de *read*.

  Testamos se o arquivo "*teste"* é nulo. Se for, houve um erro na abertura do arquivo e o usuário será informado disso e o sistema encerra.

  Fazemos o primeiro *fscanf* que fará a leitura da primeira linha. Para essa função passamos como parâmetro "*teste*" que indica o arquivo, o "%" com o tipo de variável que vai ser lida e o endereço de onde ela será armazenada.

  Na primeira linha nos foi indicado que ficaria o tamanho do vetor, então já podemos chamar a função do "*TAD_Lista" incializa_lista*.

  Fazemos o segundo *scanf* para lermos a segunda linha que indica quantas linhas de operação teremos e o armazenamos na variável "*Nlo"*. Iniciamos uma estrutura de repetição que repetirá "*Nlo"* vezes o processo de ler a próxima linha e armazenar a opção de ação e a quantidade de vezes que será repetida. Se a operação for 0 marcamos o tempo na variável "*t1"* e repetimos "*Qt*" vezes a ação de criar um item e inseri-lo na lista, através das funções apropriadas. Assim que termina o processo, "*t1"* recebe o tempo atual menos o "*t1"* anterior, calculando o tempo de execução das inserções pedidas que será armazenado na variável "*tempo_execução_inserção.* Se a operação for 1 marcamos o tempo na variável "*t2"* e repetimos "*Qt*" vezes a ação de retirar o primeiro item da lista, através da função apropriada. Assim que termina o processo "*t2"* recebe o tempo atual menos o "*t2"* anterior, calculando o tempo de execução das exclusões pedidas que será armazenado na variável "*tempo_execução_remocao".*

  Assim que saimos do *for* do "*Nlo"* fazemos um último *scanf* que lerá os números de teste (inserimos essa parte em todos nossos arquivos de teste).

Para a saída abrimos o “*arquivo_de_saida*” na variável “*saída*” com o segundo parâmetro “*a*” que permite inserir itens no final do arquivo, que caso não exista será criado. Usamos o *fprint* com os números de teste e tempo de execução em segundos. Para terminar fechamos os arquivos e damos um *break* para sair do *case* 1.

4. Resultados

Os dados obtidos pelos 12 testes pedidos no comando do Trabalho Prático estão no gráfico abaixo:

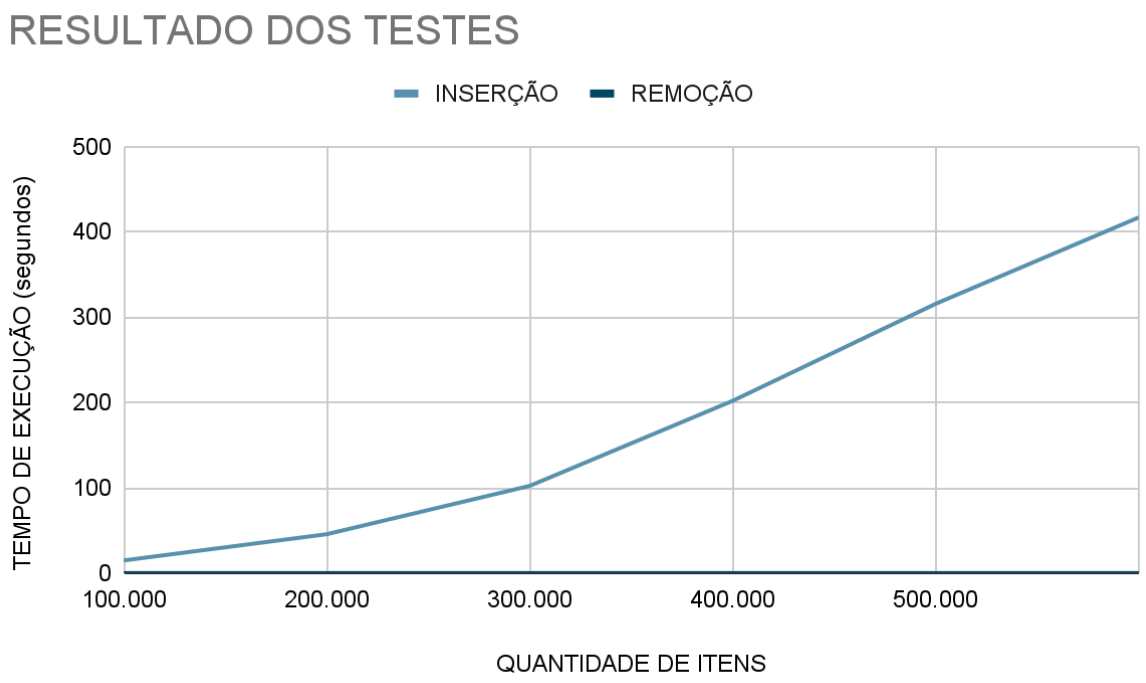

Figura 2 - Gráfico com resultado dos testes.

A partir desses dados percebemos que o processo de inserção é consideravelmente mais demorado que o processo de remoção. Tanto que nem é visível seu crescimento no gráfico já que seu tempo de execução não passa de 1 segundo. Então para analisarmos os dados da remoção teremos a tabela abaixo:

QUANTIDADE DE ITENS	INSERÇÃO	REMOÇÃO
100,000	14.879	0.003
200,000	45.617	0.005
300,000	102.258	0.015
400,000	201.982	0.009
500,000	315.725	0.029
600,000	416.873	0.019

Figura 3 - Tabela com resultado dos testes.

A tabela mostra que no geral o tempo de execução da remoção aumenta de acordo com a quantidade de itens, porém duas vezes há uma diminuição sem causa aparente.

5. Conclusão

O ponto chave da implementação do trabalho foi o uso de cursores, uma forma de encadeamento que não havíamos trabalhado anteriormente. Mas que facilita a implementação, comparado com apontadores em que as ferramentas são ponteiros ao invés de inteiros.

Uma das formas de facilitarmos a ordenação foi impedirmos a repetição de *PID.*

Já quanto aos resultados, acreditamos que a diferença tão grande entre o tempo de execução da inserção e remoção deriva-se da quantidade de comandos que cada função necessita.

6. Referências

[1] Ziviani, N. Projeto de Algoritmos. Último acesso em: 15 de dezembro de 2021.

[2] Canal Programação Descomplicada Linguagem C (Playlist: “Linguagem C: arquivos).Disponível em:

https://www.youtube.com/watch?v=LNu-0bzxpos&list=PL8iN9FQ7_jt4TfE02fK7gCkYOzL4htaE2 . Último acesso em: 20 de dezembro de 2021.