Simulador de elevador

Estrutura de Dados – turma E – Professor: George Teodoro

Autores:

Bruno Fernandes Carvalho - 150007159

Leonardo Nunes Cornelio Rêgo - 150015046

Introdução

O objetivo desse trabalho é fazer um algoritmo em C para simular um elevador sujeito a dois eventos: alguém chama o elevador ou é definido algum destino para ele. Antes da simulação, é determinado o ambiente do elevador (capacidade máxima) e o número de andares do prédio, além de um arquivo contendo os dados de entrada. Foi utilizado uma escala de tempo Zepslon, incrementada a cada andar que o elevador passa, além dele ser incrementado quando uma ou mais pessoas entram ou saem em determinado andar. Após a saída do elevador, mostra-se quanto tempo a pessoa esperou para entrar no elevador e quanto tempo ela ficou dentro dele até seu destino.

Para simular o elevador baseado em eventos reais, foi usado o seguinte método de escalonamento de processos: o elevador vai priorizar suas ações conforme a direção de deslocamento dele, ou seja, se ele está subindo ou descendo. Por exemplo, se o elevador está subindo e alguém o chama num andar acima do que ele está, ele priorizará essa ação. Caso a chamada seja num andar contrário ao sentido de movimento do elevador, ela será postergada até o momento que o elevador trocar de direção, ou seja, quando não tiver que realizar mais nenhuma ação na direção que ele está movendo. Além disso, as ações a serem feitas no mesmo sentido do elevador são ordenadas conforme a menor distância. Essa lógica representa bem uma situação real e reage de forma inteligente às ações requisitadas.

Como executar o programa

Para executar o programa, foi usado o utilitário Make, já instalado no sistema operacional Linux. Como foram utilizados quatro módulos, foi necessário criar quatro objetos (arquivo .o) para relacioná-los e criar um executável chamado “main”. A seguir, mostra-se como deve ser gerado o arquivo makefile para a correta compilação dos módulos.

```
main: main.o logica.o in_out.o lista.o
    gcc logica.o main.o in_out.o lista.o -o main
main.o: main.c logica.h lista.h
    gcc -g -c main.c
logica.o: logica.c logica.h
    gcc -g -c logica.c
in_out.o: in_out.c in_out.h
    gcc -g -c in_out.c
lista.o: lista.c lista.h
    gcc -g -c lista.c
```

Como foi usado no programa a função getopt para receber argumentos da linha de comando, é necessário definir todos eles na hora de executar a “main”. A seguir, é mostrado o menu de ajuda para executar o programa.

```
[uso]./main <opcoes>
-n Nome do arquivo   Nome do arquivo que descreve a sequência de eventos.
-a Numero de andares    configura o numero de andares.
-c Carga maxima do elevador   configura o numero maximo de pessoas no elevador.


 DIGITE OS TRES ARGUMENTOS PARA INICIAR O SIMULADOR
```

Assim, na linha de comando, deve ser escrito para executar:

./main -c 10 -a 20 -n teste.txt

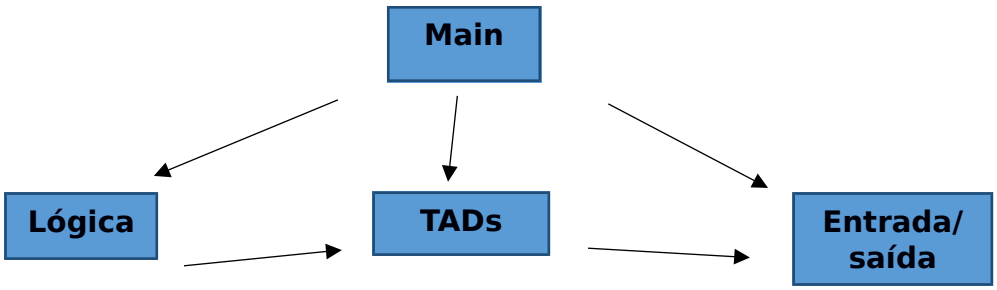
10 é o número de pessoas máxima no elevador, 20 o número de andares e teste.txt o arquivo texto contendo os dados de entrada com o eventos. O arquivo de entrada deve seguir a formatação: <andar de chamada>, <andar de destino>, <tempo de chamada>, cada um desses dados numa linha do arquivo, conforme mostrado abaixo.

```
10 ( andar de chamada )
19 ( andar de destino )
1 ( tempo de chamada )
```

Para a execução correta do simulador, é necessário que essa formatação seja seguida à risca, pois a entrada de dados depende dela. Além disso, espera-se que a entrada de dados do arquivo texto seja coerente com o número máximo de andares definido na hora da execução, pois não é testado o caso em que um andar não existe no prédio. Também se espera que uma pessoa não entre no elevador para sair no mesmo andar, já que se trata de uma análise de dados baseada em eventos reais.

Descrição dos módulos

O algoritmo foi dividido em quatro módulos: um que contém o programa principal (main), um que contém a lógica do simulador, um que gerencia a entrada e saída de dados e o último que contém os TADs (tipo abstrato de dados) e as estruturas de dados utilizadas. O programa principal depende de todos os módulos, além destes serem interdependentes, como mostrado na figura a seguir:



-Programa principal (main)

A main é responsável por articular as chamadas da maioria das funções para o funcionamento correto do simulador. Nela, também é usado a primitiva getopt para receber argumentos da linha de comando do terminal. Essa função recebe o nome do arquivo texto, o número de andares do prédio e a capacidade máxima do elevador. Esse módulo também é responsável por declarar as estruturas de dados usadas,

chamar as funções que gerenciam a entrada de dados e iniciar a simulação, controlando a chamadas das rotinas feitas na lógica do programa.

-Lógica do simulador

Essa parte do programa contém as funções responsáveis por dar vida ao simulador. Nela está toda a lógica que define o escalonamento de processos e define como a as estruturas de dados são ordenadas e utilizadas, além de atribuir a direção do elevador, verificar se o andar atual é o andar de destino ou chamada de alguém, organizar a lista de ações conforme o escalonamento utilizado, verificar no momento Zepslon se houve alguma chamada, verificar se a direção do elevador mudou e mover o elevador. Esse módulo também é responsável por atribuir os tempos de entrada e saída do elevador de cada pessoa e gerencia a saída de dados conforme o atendimento foi feito, liberando a memória. O cabeçalho das funções é mostrado a seguir:

void verifica_andar(tipolista *lista, tipolista *listaacao, tipoelevador *elevador, tipolista *listaposterior, FILE *fp, int *zepslon);
void define_direcao(tipoelevador *elevador, apontador aux);
void chamada(tipolista *lista, tipolista *listaacao, tipolista *listaposterior, int *zepslon, tipoelevador *elevador);
void organiza_listaacao(tipoelevador *elevador, tipolista *lista, tipolista *listaacao, tipolista *listaposterior, char ent_ou_saida);
void move_elevador(tipoelevador *elevador);
void verifica_direcao(tipoelevador *elevador, tipolista *listaacao, tipolista *listaposterior, tipolista *lista, int *simulação, int *muda_andar);

-Entrada e saída de dados

Esse módulo é responsável somente por organizar a entrada e saída de dados, além de mostrar o menu de ajuda relacionado com a função getopt. A função de entrada atribui a cada três linhas do arquivo texto à célula correspondente da estrutura de dados, atribuindo o andar de chamada, andar de destino e tempo de chamada. Após isso, esses dados são ordenados em ordem crescente de tempo de chamada. A função de saída apenas mostra os dados tempo de espera e tempo no elevador correspondente a cada pessoa num arquivo texto (saida.txt). Nesse arquivo, os dados são mostrados em ordem de atendimento, ou seja, em ordem crescente do tempo de chamada. O cabeçalho das funções é mostrado abaixo:

int entrada(char arq_entrada[], tipolista *lista);
void saida(FILE *fp, apontador aux);
void mostra_ajuda(char *name);

-TADs e estrutura de dados

Nessa parte do algoritmo estão os tipos abstratos de dados (especificação matemática que determina a implementação das estruturas de dados) e as próprias estruturas. Três tipos são criados: tipocelula contendo os elementos da célula e um apontador para a próxima célula da lista, tipolista que define a lista e tipoelevador, contendo os dados do elevador. Além disso, foi criado um tipo apontador, que aponta para as células criadas. Essas estruturas são mostradas a seguir:

```
typedef struct tipocelula *apontador;

typedef struct tipocelula{
   int andar_chamada, andar_destino, andar_dest,
tempo_chamada;
   int tempo_entrada, tempo_saida;
   apontador prox;
 }celula;

typedef struct{
    apontador primeiro, ultimo;
}tipolista;

typedef struct{
    int andar_atual, andar_max, cap_max, cap_atual;
    char direcao;
}tipoelevador;
```

Além disso, diversas operações matemáticas são feitas com essas estruturas, também descritas nesse módulo. Entre elas, pode-se citar ordenação da lista com os tempos de chamada, inserção, criação de uma lista vazia, desalocação, transformar a lista posterior em lista ação, inserir célula nova na lista ação, inserir nova célula na lista posterior e ordenar a lista ação de forma crescente ou decrescente, como mostrado abaixo:

```
void flvazia(tipolista *lista);
int insere(FILE *fp, tipolista *lista);
void insere_listaposterior(tipolista *listaposterior, apontador aux);
void transforma_lista(tipolista *listaacao, tipolista *listaposterior);
void insereacao(tipolista *lista,tipolista *listaacao);
void ordena_crescente(tipoelevador *elevador, tipolista *listaacao, tipolista *listaposterior);
void ordena_decrescente(tipoelevador *elevador, tipolista *listaacao, tipolista
*listaposterior);
void ordena_tempo(tipolista *lista);
apontador libera(apontador anterior, apontador aux1);
void desaloca(tipolista *lista);
```

Análise dos algoritmos e complexidade

Cada módulo possui rotinas que apresentam papel fundamental na execução correta do programa, e serão analisados os algoritmos com o objetivo de entender as escolhas feitas, entender como eles funcionam e seus pontos fortes e fracos. Além disso, será analisado a complexidade dessas funções para determinar a otimização do código.

-Programa principal (main)

Na main, é preciso que a simulação aconteça enquanto houver dados de entrada a serem lidos e todas as ações do elevador tenham sido executadas. O loop que mantém a simulação rodando é descrito a seguir.

```
while (simulacao == 1)

        {
        chamada(&lista, &listaacao, &listaposterior, &zepslon, &elevador);
        verifica_andar(&lista, &listaacao, &elevador, &listaposterior, fp, &zepslon);
        verifica_direcao(&elevador, &listaacao, &listaposterior, &lista, &simulacao);
        if (muda_andar == 0){
                move_elevador(&elevador);
        }
        zepslon++;
        muda_andar = 0;
        }
```

A cada loop, é testado se houve alguma chamada e se o andar atual do elevador é o andar de chamada ou destino de alguém. Verifica-se também a direção do elevador e a muda caso necessário e move-se o elevador dependendo da sua direção de movimento. No final, o tempo é incrementado. Percebe-se que é um algoritmo simples que corresponde à sua funcionalidade: simular enquanto houver dados e ações a serem executadas. Quando essas condições acabarem, a simulação acaba.

Análise de complexidade

Como dito acima, enquanto houver dados para serem processados, o programa continuará no loop, porém o número de vezes que o loop é executado não depende do número de eventos(cada passageiro é um evento), e sim, do último tempo de chamada. Por exemplo, se um teste contém apenas um passageiro, e o tempo que ele chama o elevador é 600 Zepslons, o loop será rodado no mínimo 600 vezes até realizar a ação desejada. Seja “i” o tempo da última chamada(em zepslon) e sabendo

que cada atribuição dentro do loop é O(1), a complexidade desse trecho é O(i).

-Lógica do simulador

Apesar de já ter sido descrita a técnica de escalonamento, ela será discutida a fundo nesse tópico. A lógica do simulador baseia-se numa simulação real, em que o elevador obedece primeiro as ações no sentido em que se move na ordem de menor esforço, e depois executa as do sentido contrário. Acredita-se que a escolha dessa técnica foi a melhor possível para resolver o problema. Comparando com duas técnicas encontradas na literatura, percebe-se que FIFO (First in, First out) faz o elevador mover-se desnecessariamente para cumprir suas ações, e SJF (Shortest Job First) apresenta problemas que dificultam uma simulação real. Um desses problemas é que como ele sempre vai realizar o trabalho de menor esforço, pode ser que alguma chamada demore muito para ser executada, tornado a simulação desotimizada. Assim, o algoritmo implementado resolve o problema de forma otimizada, escolhendo as ações de forma que o elevador não mova desnecessariamente e nem demore muito para atender os pedidos.

Para estabelecer um algoritmo que siga essa lógica, foi necessário criar três listas: uma lista ordenada em ordem de tempo de chamada, uma lista de ações recentes do elevador e uma lista posterior, que contém as ações que serão executadas posteriormente. Assim, a cada tempo Zepslon, verifica-se na primeira lista se houve alguma chamada, e caso tenha, é necessário organizar a lista de ações do elevador. Se ele tiver subindo, todas as ações que estiverem fazendo o elevador continuar no seu sentido são ordenadas em ordem crescente. As ações que requisitarem o elevador num andar abaixo do que ele está serão colocadas na lista de ações posterior, e serão transferidas para a lista de ações recentes do elevador quando ele trocar de direção. Nesse exemplo, depois que o elevador trocar de direção, as ações serão ordenadas de forma decrescente.

Além dessas funcionalidades, o algoritmo é inteligente o suficiente para testar se o elevador está lotado, e caso esteja, coloca essa pessoa na lista de ações posterior. Outro ponto forte é que quando mais de uma pessoa quer entrar e sair no mesmo andar, primeiro considera-se que as pessoas saem para esvaziar o elevador, e depois testa-se a capacidade deste para ver se as pessoas podem entrar.

Dessa forma, é possível escalonar as ações seguindo a lógica descrita anteriormente. Apesar da inteligência desse método, alguns

pontos fracos aparecem: toda vez que uma pessoa chama o elevador, é necessário inserir esta na lista de ações, e posteriormente, ordenar a lista, seguindo as condições descritas acima. Assim, em vez de apenas inserir a pessoa no local certo da lista de ações, optou-se por toda vez ordenar a lista. Além disso, a cada andar é feito um teste para ver se o elevador alcançou sua primeira ação, resultando em testes desnecessários muitas vezes.

Análise de complexidade das principais funções

A maior parte das funções da lógica baseiam-se em comparações e sem estruturas mais complexas como os loops. A maior parte da funções são 0(1), pois simplesmente fazem testes e chamam funções mais complexas para resolver os problemas. Existem duas funções mais complexas. Na função, “verificar_andar”, ela testa todos os elementos que estão na lista de ações, ou seja, as ações que possuem prioridade. Por esse motivo é utilizado um loop para testar se algum pessoa deve sair ou entrar no elevador. Seja “i” o número de ações a serem realizadas pelos elevador no momento, o loop será percorrido i vezes, portanto essa função é O(i). A segunda função, “chamada”, verifica a quantidade de chamadas no instante zepslon. Considerando “i” o número de chamadas em um determinado zepslon, o loop será percorrido i vezes, portanto essa função é O(i).

-Entrada e saída de dados

O algoritmo de entrada e saída de dados é bastante simples, contendo apenas funções responsáveis por receber os dados de entrada, imprimir os dados de saída e o menu de ajuda para executar corretamente o programa pelo terminal.

Os dados de saída são mostrados em um arquivo, denominado “saida.txt”. Neste, dois dados são impressos por ordem de atendimento: um é o tempo que a pessoa espera até que o elevador chegue no andar de chamada depois que ela o chama, e o outro dado é o tempo que a pessoa fica dentro do elevador até que o elevador chegue no andar de destino, ambos em zepslons.

Como as funções desse modulo quase não apresentam complexidade, acredita-se que a implementação delas foi muito boa, resultando numa boa otimização do módulo.

Análise de complexidade

No início do programa, todos os eventos do arquivo texto serão passados para uma lista. Seja “n” o número de eventos, o loop da função que realiza a entrada de dados será percorrido n vezes, e como os

comandos dentro do loop são apenas atribuições(que são O(1)), a complexidade da função é O(n). Já para a função de saída, é realizado apenas um print no arquivo, portanto O(1). Vale lembrar que essa função será executada n vezes no programa.

-TADs e estrutura de dados

O algoritmo descrito nesse módulo é responsável por dar suporte à lógica do simulador. Aqui são definidas diversas funções que fazem operações com os TADs, além do próprio tipo e das estruturas que estão contidas nele. Escolheu-se usar listas simples usando ponteiros como TAD, já que todas as listas são alocadas dinamicamente e não se sabe o tamanho máximo dela. Outro ponto forte dessa escolha é que é criado apenas uma célula por evento no programa, e quando deseja-se mudar essa célula para outra lista, são feitas apenas operações com ponteiros, não sendo necessário ocupar mais espaço na memória. Quando a pessoa é atendida e chega no seu andar de destino, o espaço que ela ocupa no programa é liberado, resultando num código dinâmico e eficiente.

Existem várias funções que operam sobre as listas criadas, já comentadas na descrição dos módulos. As funções de inserção basicamente adicionam a célula à lista desejada, e as funções de liberação são responsáveis por desalocar algum espaço de memória. Outra função de extrema importância é a que transforma a lista de ações posteriores na lista de ações recentes, mudando apenas os ponteiros. Para facilitar a manipulação das listas, preferiu-se quase sempre mexer com o elemento da primeira posição.

Apesar dessas funções serem indispensáveis para o algoritmo principal, tem outras que se destacam, como as de ordenação. A lógica utilizada para implementar isso foi ficar testando de dois em dois elementos e ir mudando a posição deles até que estejam totalmente ordenados. A função “ordena_tempo” é chamada no início do programa para ordenar os tempos de chamada de forma crescente. Já as funções “ordena_crescente” e “ordena_decrescente” apresentam uma lógica parecida, mas com uma diferença sútil. Elas são responsáveis por ordenar a lista de ações do elevador conforme o escalonamento estabelecido, então ao invés de sempre ordenar a lista, antes deve ser testado o andar atual do elevador. Na função “ordena_crescente”, por exemplo, serão ordenadas todas as ações que tem como destino um andar acima do andar atual do elevador, e as ações que estão no sentido contrário são inseridas na lista posterior.

Verifica-se que essas funções que ordenam a lista de ações do elevador são chamadas muitas vezes, e toda vez elas a ordenam, resultando em testes e processamento desnecessários em muitos casos. Além disso, o método de ordenação apresentado não é o mais otimizado, resultando num ponto negativo do algoritmo.

Análise de Complexidade

É nesse módulo que são realizadas as operações mais custosas para o nosso programa, então logicamente serão as funções com maiores complexidades. Como nosso algoritmo é baseado na ordenação constante de listas de ações, é muito recorrente a chamada de funções de ordenamento. Resolvemos implementar a ordenação mais simples, que terá uma complexidade $O(n^2)$, pois consiste em um loop dentro de outro loop, aonde toda vez que vamos fazer uma ordenação, é percorrido toda a lista, fazendo-se comparações. As outras funções são mais simples e normalmente tem complexidade O(1).

Análise de dados

Foram executados alguns testes para verificar a corretude do algoritmo e serão analisados em relação ao número de eventos, tempo de execução e memória gasta.

As condições iniciais do elevador para os 4 primeiros testes são:

- Andar inicial do elevador: 0;
- Lotação máxima: 10.
- Andar máximo: 20;

Teste 1 (10 Eventos)				
Tempo gasto: 0 ms		Memória utilizada: 3544Kbytes		
Andar de Chamada	Andar de Destino	Tempo de Chamada	Tempo de Espera	Tempo no Elevador
4	7	0	6	3
7	13	2	8	7
7	14	0	10	9

3	19	3	1	21
2	0	0	2	43
7	0	2	8	35
11	0	1	14	30
11	0	0	15	30
0	7	2	44	7
0	17	2	44	18

Teste 2 (20 Eventos)				
Tempo gasto: 0 ms		Memória utilizada: 3544Kbytes		
Andar de Chamada	Andar de Destino	Tempo de Chamada	Tempo de Espera	Tempo no Elevador
12	14	7	11	2
10	15	0	14	8
15	16	1	22	1
12	18	6	12	10
6	19	1	6	23
12	6	4	14	27
17	4	2	25	21
5	0	2	3	49
7	0	6	3	45
8	0	0	11	43
10	0	0	14	40
11	0	7	9	38
19	0	4	27	23

0	2	8	47	2
0	4	5	50	5
1	7	8	45	11
0	7	1	54	9
0	8	4	51	11
0	12	3	52	16
13	17	8	30	39

Teste 3 (30 Eventos)				
Tempo gasto: 0 ms		Memória utilizada: 3544Kbytes		
Andar de Chamada	Andar de Destino	Tempo de Chamada	Tempo de Espera	Tempo no Elevador
9	10	8	4	1
6	11	4	3	8
13	16	11	9	4
12	17	4	14	8
11	18	6	10	12
11	19	13	3	14
9	19	1	11	18
15	13	7	29	2
13	8	2	18	24
18	8	9	20	15
14	2	10	12	32
2	0	6	49	2
2	0	3	52	2
5	0	10	39	8
3	0	0	3	54

8	0	10	0	47
8	0	7	3	47
18	0	13	16	28
0	3	7	51	3
0	7	9	49	8
4	9	12	39	18
0	9	8	50	11
3	12	13	40	20
0	15	1	57	19
0	15	4	54	19
2	16	6	49	24
0	16	5	53	21
0	19	6	52	25
0	1	13	92	1
0	10	9	96	11

Teste 4 (40 Eventos)				
Tempo gasto: 0 ms		Memória utilizada: 3544Kbytes		
Andar de Chamada	Andar de Destino	Tempo de Chamada	Tempo de Espera	Tempo no Elevador
0	5	0	0	6
7	8	2	8	1
4	11	4	1	11
8	12	11	1	6
11	14	5	12	5
15	16	8	17	1

9	19	14	0	16
19	18	2	29	1
15	12	4	21	14
15	8	1	24	19
14	6	3	20	24
14	5	11	12	26
11	4	8	9	34
6	3	15	33	5
8	3	10	2	41
18	2	1	32	22
4	0	11	41	6
13	0	14	7	37
13	0	8	13	37
16	0	14	13	31
0	3	5	54	3
0	4	4	55	5
0	8	2	57	11
0	8	3	56	11
8	11	14	31	29
8	12	13	32	31
0	12	3	56	17
0	13	1	58	19
5	17	12	38	34
3	19	16	38	33
16	2	7	76	22

7	0	15	54	39
16	0	2	81	25
0	2	6	103	2
0	3	12	97	4
0	7	11	98	9
0	7	12	97	9
0	17	8	101	20
0	19	9	100	23
0	19	16	93	23

Percebe-se pela análise dos dados obtidos que o algoritmo representa uma boa simulação de um elevador real, representando tempo de espera e tempo no elevador dentro do esperado, com uma boa otimização do escalonamento de processos. Foram feitos outros testes para testar o limite do algoritmo e foram obtidos os seguintes valores:

No teste 5, foi utilizado um número de eventos(pessoas) igual a 1000 para as mesmas condições iniciais dos testes anteriores. Foram gastos 3552 Kbytes de memória e o tempo de execução foi de 126 ms. Esse valor de tempo de execução encontrado foi bem maior que os outros, pois a carga processada foi muito maior.

No teste 6, foi utilizado um número de eventos(pessoas) igual a 10.000 para condições iniciais diferentes, visto que para uma carga de 10.000 eventos, a lotação máxima do elevador deve ser aumentada. Utilizou-se uma lotação máxima de 50 pessoas e número de andares 20. Foram gastos 3552 Kbytes de memória e o tempo de execução foi de 3,09 s. Isso mostra que para uma quantidade muito elevada de eventos, o código já não se torna tão eficiente, pois seu tempo de execução começa a se tornar alto.

Conclusão

Conclue-se que os tempos de espera e os tempos dentro do elevador se tornaram os mais uniformes possíveis, ou seja, a maior partes das ações terá um tempo de realização aceitável para uma aplicação real. O método implementado possui problemas, e um deles é que utiliza-se muito processamento computacional, pois precisa-se realizar diversos ordenamentos nas listas durante o programa, e essas ações são muito caras. Num balanço final, acredita-se que a solução implementada para resolver o problema proposto foi inteligente e poderia ser aplicada em situações reais, já que não requer o processamento de muitas ações ao mesmo tempo.