

Consultório Médico

Gabriel de Castro Dias - 211055432

9 de novembro de 2025

1 Introdução

Neste relatório, abordaremos um problema desenvolvido para o trabalho da disciplina de programação concorrente da Universidade de Brasília. Será apresentada uma formalização do problema, um algoritmo que o soluciona e, por fim, como esse projeto é útil para o aprendizado e capacitação dos alunos da universidade.

2 Descrição do Problema

O problema simula o funcionamento de um consultório médico, onde teremos threads para médicos, recepcionistas e pacientes, além de semáforo para assentos e geração randômica para pacientes e outras flags. O desafio proposto será criar um código onde não haja interreferência entre as threads, tomando cuidado onde cada elemento é posicionado para evitar problemas como deadlock, starvation, condição de corrida e N outras falhas que uma programação não-determinística como a programação concorrente possa gerar, portanto será necessário o uso de variáveis de condições, locks e variáveis globais para controlar o fluxo e proteger os pontos críticos.

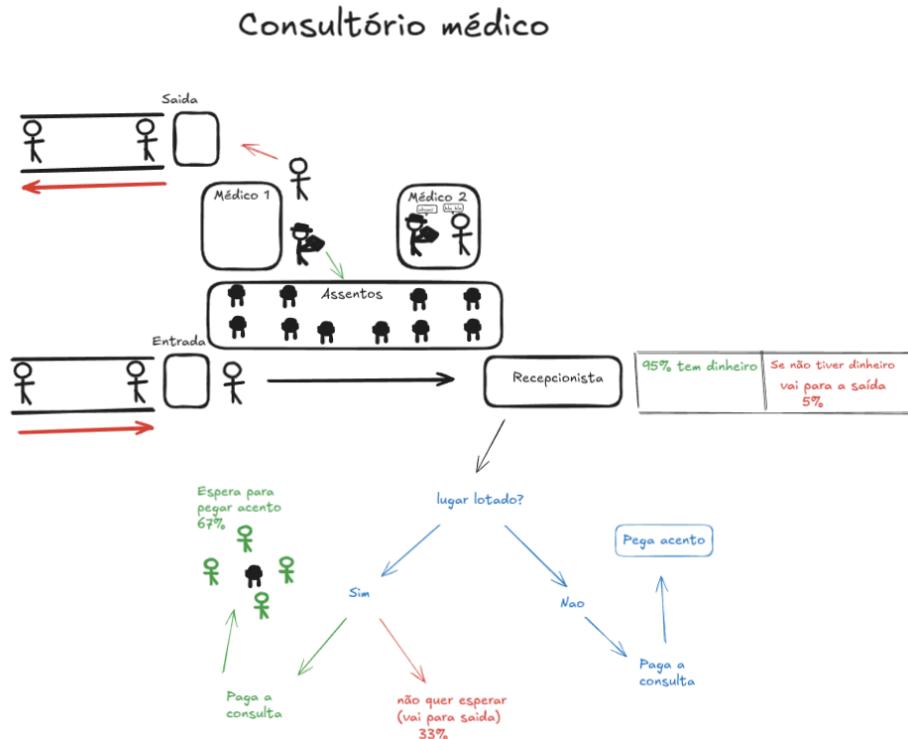


Figura 1: Imagem do Consultório.

A simulação do nosso fluxo funciona da seguinte maneira:

1. Uma pessoa entra no consultório e acorda a recepcionista.
2. Recepcionista realiza o atendimento da pessoa, pede o pagamento da consulta e informa se há assentos livres.
 - (a) Se a pessoa não tiver dinheiro (esqueceu a carteira em casa) ela não vai consultar e vai para a saída.
 - (b) Se a pessoa tiver dinheiro e assentos livres, ela realizará o pagamento da consulta, pegará um dos assentos e aguardará o atendimento do médico.
 - (c) Se a pessoa tiver dinheiro, mas não houver assentos livres, ela pode realizar o pagamento e esperar a liberação de um assento ou desistir da consulta e ir embora.
3. A pessoa sentada em um dos assentos torna-se um paciente e acorda o médico; em seguida, dorme até ser acordado para a consulta.
4. Quando o médico chama, o paciente vai até a sala e a consulta é simulada com um sleep().
5. Consulta realizada, o paciente vai embora e o médico chama o próximo paciente.
6. Assim segue o fluxo até o fim dos pacientes, que são definidos de forma aleatória entre 1 à 50 a cada execução do programa.

Além da randomização de pacientes a cada execução, também é feita a mesma coisa para flags de cada pessoa para definir se ela tem dinheiro e se vai esperar ou não. A realização dessa geração aleatória é implementada na main() do programa.

3 Algoritmo

Código implementado com os conhecimentos da matéria, ministrada pelo professor [Alchieri \(2025\)](#).

```

1 #include "stdio.h"
2 #include "unistd.h"
3 #include "stdlib.h"
4 #include "pthread.h"
5 #include "semaphore.h"
6
7 #define MEDICOS 2           //numero de medicos
8 #define ASSENTOS 10        //numero de assentos disponiveis
9
10 pthread_cond_t medico_cond = PTHREAD_COND_INITIALIZER;
11 pthread_cond_t paciente_cond = PTHREAD_COND_INITIALIZER;
12 pthread_cond_t recepcionista_cond = PTHREAD_COND_INITIALIZER;
13 pthread_cond_t atendimento_cond = PTHREAD_COND_INITIALIZER;
14 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
15 sem_t semaforo_assentos;
16
17 int quantidadePacientes = 0;
18 int pessoas = 0;
19 int lotado = 0;
20
21 typedef struct{
22     int* id;
23     int vai_esperar;
24     int tem_dinheiro;
25 } PacienteArgs;
26
27
28 void * medico(void *arg){
29     int i = *((int *) arg);      // id
30     while(1){
31         pthread_mutex_lock(&mutex);
32         while(quantidadePacientes == 0){    // enquanto nao tem paciente, dorme
33             pthread_cond_wait(&medico_cond, &mutex);
34         }
35         //acorda
36         quantidadePacientes--;

```

```

37     pthread_cond_signal(&paciente_cond);      // chama um dos pacientes
38     printf("Medico %d: Proximo paciente!\n", id);
39     pthread_mutex_unlock(&mutex);
40     sleep(5);        // realiza a consulta
41 }
42 }
43 }
44
45
46 void * paciente(void *arg){
47
48     PacienteArgs *pArgs = (PacienteArgs *) (arg); // converte void para o tipo da
estrutura
49     int id = *(pArgs->id);
50     int vai_esperar = pArgs->vai_esperar;
51     int tem_dinheiro = pArgs->tem_dinheiro;
52
53     free(pArgs);
54
55     pthread_mutex_lock(&mutex);
56     pessoas++;
57     pthread_cond_signal(&recepcao_cond);           // acorda recepcionista
58
59     //espera atendimento
60     pthread_cond_wait(&atendimento_cond, &mutex);
61     printf("Paciente %d: Fui atendido pelo recepcionista\n", id);
62
63     if(tem_dinheiro == 0){
64         printf("Paciente %d: Desculpe, esqueci o dinheiro em casa, vou tentar voltar
amanha\n", id);
65         pessoas--;
66         pthread_mutex_unlock(&mutex);
67         return NULL;
68     }
69     if(lotado != 0){ // esta lotado
70         if(vai_esperar == 0){ // nao vai esperar
71             printf("Paciente %d: Essa fila toda? aff eu que nao vou esperar!\n", id);
72             pessoas--;
73             pthread_mutex_unlock(&mutex);
74             return NULL;
75         }
76         printf("Paciente %d: Que fila ein... vou ter que esperar, fazer o que.\n", id)
77     }
78
79     printf("Paciente %d: Pagamento concluido!\n", id);
80
81     pessoas--;
82
83     pthread_mutex_unlock(&mutex);
84
85     printf("Paciente %d: Tentando pegar um assento...\n", id);
86     sem_wait(&semaforo_assentos);                // pega assento
87     printf("Paciente %d: Conseguir um assento. \n", id);
88
89     pthread_mutex_lock(&mutex);
90     quantidadePacientes++;
91     pthread_cond_signal(&medico_cond);           // acorda um dos medicos
92     pthread_cond_wait(&paciente_cond, &mutex);    // dorme esperando o medico chamar
93     // medico chamou
94     printf("Paciente %d: O medico me chamou.\n", id);
95     sem_post(&semaforo_assentos);                 // libera assento
96
97     pthread_mutex_unlock(&mutex);
98
99     printf("Paciente %d consultando com o medico\n", id);
100    sleep(2);
101    printf("Paciente %d terminou a consulta e foi para casa\n", id);
102    return NULL;
103 }
104

```

```

105 void * recepcionista(void *arg){
106     while(1){
107         pthread_mutex_lock(&mutex);
108         while(pessoas == 0){ // se nao tiver pessoas, dorme
109             pthread_cond_wait(&recepcao_cond, &mutex);
110         }
111         // pessoa acorda recepcionista
112         printf("Recepção: Olá, seja bem vindo(a), documentos por favor\n");
113         printf("Recepção: Analisando os documentos da pessoa\n");
114         printf("Recepção: Por favor realize o pagamento no valor de R$150,00 para
o nosso CNPJ.\n");
115
116         //confere se assentos estão lotados
117         int valor_assentos;
118         sem_getvalue(&semaforo_assentos, &valor_assentos);
119
120         if(valor_assentos != 0){ // tem assentos
121             lotado = 0;
122             printf("Temos assentos disponíveis\n");
123         }
124         else{
125             lotado = 1;
126             printf("Infelizmente estamos sem vagas nos assentos, porém se alguém sair
voce pode tentar pegar.\n");
127         }
128
129         pthread_cond_signal(&atendimento_cond); // avisa que o atendimento acabou
130
131         pthread_mutex_unlock(&mutex);
132
133         sleep(1);
134     }
135
136
137 }
138
139
140 int main() {
141     int i;
142     int* id;
143     int vai_esperar;
144     int tem_dinheiro;
145
146     // inicializa semaforo
147     sem_init(&semaforo_assentos, 0, ASSENTOS);
148
149     srand(time(NULL)); // inicializa gerador aleatório
150     int randomNumber;
151     int numero_pacientes = rand() % 50; // quantidade aleatória de pacientes entre 0
a 49
152
153     printf("Quantidade de pacientes hoje: %d\n", numero_pacientes);
154
155     pthread_t m[MEDICOS], p[numero_pacientes], r;
156
157     pthread_create(&r, NULL, recepcionista, NULL); // criando recepcionista
158
159     /* criando médicos */
160     for (i = 0; i < MEDICOS ; i++) {
161         id = (int *) malloc(sizeof(int));
162         *id = i;
163
164         if(pthread_create(&m[i], NULL, medico, (void *) (id)) != 0){
165             perror("Falha ao criar Thread de Médico!");
166             exit(EXIT_FAILURE);
167         }
168     }
169
170
171     /* criando pacientes */
172     for (i = 0; i < numero_pacientes; i++) {

```

```

173     id = (int *) malloc(sizeof(int));
174     *id = i;
175
176     // comecam com sim (vai esperar e tem dinheiro)
177     vai_esperar = 1;
178     tem_dinheiro = 1;
179
180     //calcula se paciente vai esperar na fila
181     randomNumber = rand() % 3; // gera entre 0 a 2      -> 1/3 = 33%
182     if(randomNumber == 2){ // escolhi um dos numeros para decidir se ele NAO vai
183         esperar
184             vai_esperar = 0;
185     }
186
187     //calcula se paciente tem dinheiro
188     randomNumber = rand() % 20; // gera entre 0 a 19    -> 1/20 = 5%
189     if(randomNumber == 4){ // escolhi um dos numeros para nao ter dinheiro
190         tem_dinheiro = 0;
191     }
192
193     PacienteArgs *pacienteArgs = (PacienteArgs *) malloc(sizeof(PacienteArgs));
194     if(pacienteArgs == NULL){
195         perror("Falha ao alocar memoria para pacienteArgs!");
196         exit(EXIT_FAILURE);
197     }
198     pacienteArgs->id = id;
199     pacienteArgs->vai_esperar = vai_esperar;
200     pacienteArgs->tem_dinheiro = tem_dinheiro;
201
202     if(pthread_create(&p[i], NULL, paciente, (void *) (pacienteArgs)) != 0){
203         perror("Falha ao criar Thread de Paciente!");
204         exit(EXIT_FAILURE);
205     }
206
207
208     for(i = 0; i < numero_pacientes; i++){
209         pthread_join(p[i],NULL);
210     }
211
212     return 0; // Medico morreu de trabalhar e Repcionista morreu de fome
213 }
```

Listing 1: Código fonte da solução.

Para rodar o programa não esqueça de compilar o código usando a flag `-pthread`.

4 Conclusão

Apesar de poucas entidades e um objetivo claro e bem definido, ainda assim foi um quebra cabeça solucionar esse problema, houveram vários casos onde a lógica na cabeça fazia sentido mas o programa não entregava o resultado esperado, casos difíceis de perceber ocorreram várias vezes, como por exemplo uma thread tentar pegar um semáforo com um wait ao mesmo tempo que tinha em posse o lock, assim travando todas as outras threads quando não haviam assentos disponíveis. Um projeto desse tipo ajuda os alunos a compreenderem melhor os desafios da concorrência, além disso deixar um escopo aberto para a criação dos nossos próprios problemas incentiva a criatividade e gera problemas diversos dependendo de cada aluno.

Referências

Eduardo Adilio Pelinson Alchieri. Aulas, slides e códigos. Material da disciplina de Programação Concorrente. Plataforma Aprender3, 2025. Acesso restrito a alunos da disciplina.