

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 interferência 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, portando será necessário o uso de variáveis de condições, locks e variáveis globais para controlar o fluxo e proteger os pontos críticos.

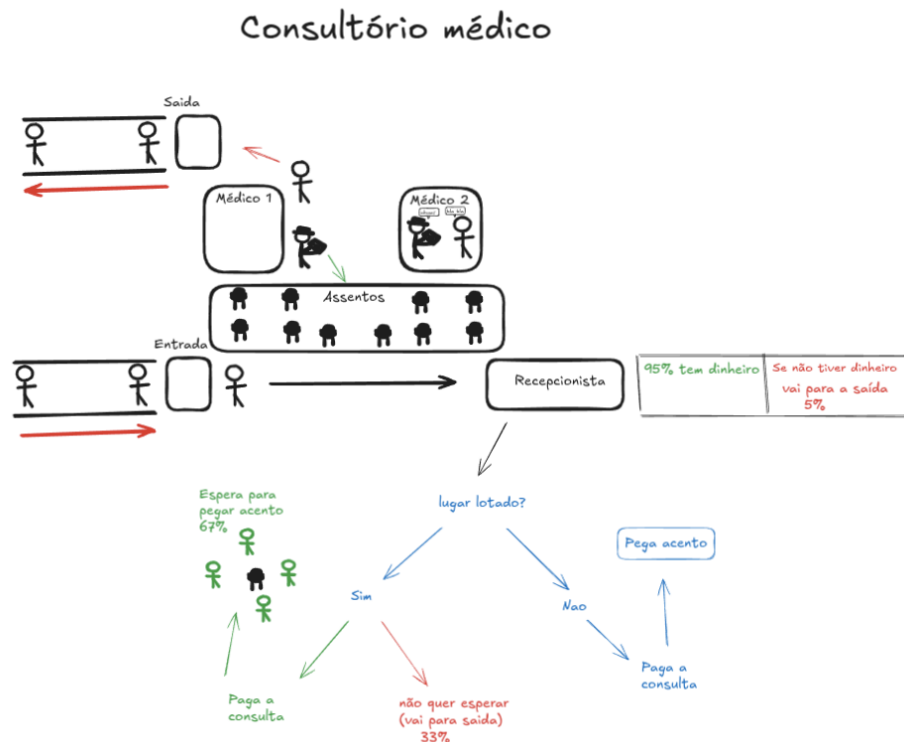


Figura 1: Imagem do Consultório.

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 void * medico(void *arg){
28     int i = *((int *) arg);    // id
29     while(1){
30         pthread_mutex_lock(&mutex);
31         while(quantidadePacientes == 0){    // enquanto nao tem paciente, dorme
32             pthread_cond_wait(&medico_cond, &mutex);
33         }
34         //acorda
35         quantidadePacientes--;

```

```

37
38     pthread_cond_signal(&paciente_cond); // chama um dos pacientes
39     printf("Medico %d: Proximo paciente!\n", i);
40     pthread_mutex_unlock(&mutex);
41     sleep(5); // realiza a consulta
42 }
43 }
44
45
46 void * paciente(void *arg){
47
48     PacienteArgs *pArgs = (PacienteArgs *) (arg); // converte void para o tipo da
49     estrutura
50     int id = *(pArgs->id);
51     int vai_esperar = pArgs->vai_esperar;
52     int tem_dinheiro = pArgs->tem_dinheiro;
53
54     free(pArgs);
55
56     pthread_mutex_lock(&mutex);
57     pessoas++;
58     pthread_cond_signal(&recepcionista_cond); // acorda recepcionista
59
60     //espera atendimento
61     pthread_cond_wait(&atendimento_cond, &mutex);
62     printf("Paciente %d: Fui atendido pelo recepcionista\n", id);
63
64     if(tem_dinheiro == 0){
65         printf("Paciente %d: Desculpe, esqueci o dinheiro em casa, vou tentar voltar
66         amanhã\n", id);
67         pessoas--;
68         pthread_mutex_unlock(&mutex);
69         return NULL;
70     }
71     if(lotado != 0){ // esta lotado
72         if(vai_esperar == 0){ // nao vai esperar
73             printf("Paciente %d: Essa fila toda? aff eu que nao vou esperar!\n", id);
74             pessoas--;
75             pthread_mutex_unlock(&mutex);
76             return NULL;
77         }
78         printf("Paciente %d: Que fila ein... vou ter que esperar, fazer o que.\n", id);
79
80         ;
81     }
82
83     printf("Paciente %d: Pagamento concluido!\n", id);
84
85     pessoas--;
86
87     pthread_mutex_unlock(&mutex);
88
89     printf("Paciente %d: Tentando pegar um assento...\n", id);
90     sem_wait(&semaforo_assentos); // pega assento
91     printf("Paciente %d: Consegui um assento. \n", id);
92
93     pthread_mutex_lock(&mutex);
94     quantidadePacientes++;
95     pthread_cond_signal(&medico_cond); // acorda um dos medicos
96     pthread_cond_wait(&paciente_cond, &mutex); // dorme esperando o medico chamar
97     // medico chamou
98     printf("Paciente %d: O medico me chamou.\n", id);
99     sem_post(&semaforo_assentos); // libera assento
100
101     pthread_mutex_unlock(&mutex);
102
103     printf("Paciente %d consultando com o medico\n", id);
104     sleep(2);
105     printf("Paciente %d terminou a consulta e foi para casa\n", id);
106     return NULL;
107 }
108 }
109
110

```

```

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(&recepcionista_cond, &mutex);
110         }
111         // pessoa acorda recepcionista
112         printf("Recepcionista: Ola, seja bem vindo(a), documentos por favor\n");
113         printf("Recepcionista: Analisando os documentos da pessoa\n");
114         printf("Recepcionista: Por favor realize o pagamento no valor de RS150,00 para
o nosso CNPJ.\n");
115
116         //confere se assentos estao 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 disponiveis\n");
123         }
124         else{
125             lotado = 1;
126             printf("Infelizmente estamos sem vagas nos assentos, porem se alguem 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 aleatorio
150     int randomNumber;
151     int numero_pacientes = rand() % 50; // quantidade aleatoria 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 medicos */
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 Medico!");
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
esperar
183         vai_esperar = 0;
184     }
185
186     //calcula se paciente tem dinheiro
187     randomNumber = rand() % 20; // gera entre 0 a 19    -> 1/20 = 5%
188     if(randomNumber == 4){ // escolhi um dos numeros para nao ter dinheiro
189         tem_dinheiro = 0;
190     }
191
192     PacienteArgs *pacienteArgs = (PacienteArgs *) malloc(sizeof(PacienteArgs));
193     if(pacienteArgs == NULL){
194         perror("Falha ao alocar memoria para pacienteArgs!");
195         exit(EXIT_FAILURE);
196     }
197     pacienteArgs->id = id;
198     pacienteArgs->vai_esperar = vai_esperar;
199     pacienteArgs->tem_dinheiro = tem_dinheiro;
200
201     if(pthread_create(&p[i], NULL, paciente, (void *) (pacienteArgs)) != 0){
202         perror("Falha ao criar Thread de Paciente!");
203         exit(EXIT_FAILURE);
204     }
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 Recepcionista 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.