

## Integrantes


Davi Micale – 32096062

Victor Ferracini – 32273134

Yuri Nichimura - 32230877

# Relatório Projeto 1 – Controlando concorrência de contas bancárias

## Construção do código no AWS:

AWS  Used \$1.7 of \$100 03:57 ▶ Stop

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <sched.h>
#include <stdio.h>
#include <pthread.h> // Adicionamos a biblioteca pthread para usar mutex
#include <time.h> // Adicionamos a biblioteca time para inicializar a semente aleatória

// 64kB stack
#define FIBER_STACK (1024 * 64)

struct c {
    int saldo;
};

typedef struct c conta;
conta contas[2]; // Alteramos para um array de contas
int valor;

// Mutex para controlar o acesso às contas
pthread_mutex_t mutex;

// Função de transferência
void* transferencia(void* arg)
{
    int i;
    for (i = 0; i < 10; i++) {
        // Escolhe contas aleatoriamente
        int conta_debitar = rand() % 2;
        int conta_creditar = 1 - conta_debitar;

        pthread_mutex_lock(&mutex); // Bloqueia o acesso às contas
        if (contas[conta_debitar].saldo >= valor) {
            contas[conta_debitar].saldo -= valor;
            contas[conta_creditar].saldo += valor;
        }
        pthread_mutex_unlock(&mutex); // Libera o acesso às contas

        // Constrói as mensagens
        char msg_saldo_conta0[100];
        char msg_saldo_conta1[100];
        char msg_transferencia[100];

        snprintf(msg_saldo_conta0, sizeof(msg_saldo_conta0), "Saldo da conta 0: %d\n", contas[0].saldo);
        snprintf(msg_saldo_conta1, sizeof(msg_saldo_conta1), "Saldo da conta 1: %d\n", contas[1].saldo);
    }
}
```

```
return NULL;
}

int main()
{
    void* stack;
    pid_t pid;
    int i;

    // Inicializa o mutex
    pthread_mutex_init(&mutex, NULL);

    // Inicializa a semente aleatória
    srand(time(NULL));

    // Aloca o stack
    stack = malloc(FIBER_STACK);
    if (stack == 0) {
        perror("malloc: could not allocate stack");
        exit(1);
    }

    // Inicializa as contas com saldo 100
    for (i = 0; i < 2; i++) {
        contas[i].saldo = 100;
    }

    printf("Transferindo 10 entre contas aleatórias\n");
    valor = 10;

    // Cria 100 threads para realizar as transferências
    pthread_t threads[100];
    for (i = 0; i < 100; i++) {
        if (pthread_create(&threads[i], NULL, transferencia, NULL) != 0) {
            perror("pthread_create");
            exit(2);
        }
    }

    // Espera que todas as threads terminem
    for (i = 0; i < 100; i++) {
        pthread_join(threads[i], NULL);
    }

    // Libera o stack
    free(stack);

    // Destrói o mutex
```

```
pid_t pid;
int i;

// Inicializa o mutex
pthread_mutex_init(&mutex, NULL);

// Inicializa a semente aleatória
srand(time(NULL));

// Aloca o stack
stack = malloc(FIBER_STACK);
if (stack == 0) {
    perror("malloc: could not allocate stack");
    exit(1);
}

// Inicializa as contas com saldo 100
for (i = 0; i < 2; i++) {
    contas[i].saldo = 100;
}

printf("Transferindo 10 entre contas aleatórias\n");
valor = 10;

// Cria 100 threads para realizar as transferências
pthread_t threads[100];
for (i = 0; i < 100; i++) {
    if (pthread_create(&threads[i], NULL, transferencia, NULL) != 0) {
        perror("pthread_create");
        exit(2);
    }
}

// Espera que todas as threads terminem
for (i = 0; i < 100; i++) {
    pthread_join(threads[i], NULL);
}

// Libera o stack
free(stack);

// Destrói o mutex
pthread_mutex_destroy(&mutex);

printf("Transferências concluídas e memória liberada.\n");
return 0;
}
```

## Lea

[Environ](#)  
[Environ](#)  
[Access t](#)  
[Region I](#)  
[Service](#)  
[Using th](#)  
[Running](#)  
[Using th](#)  
[Preserv](#)  
[Accessir](#)  
[SSH Acc](#)  
[SSH Acc](#)  
[SSH Acc](#)

Instructi

## Envir

This Lea  
environ  
services

This en

**Passo a passo do código:**

Nesta parte, são incluídas as bibliotecas necessárias para o programa, incluindo `stdlib.h`, `pthread.h`, `stdio.h`, `time.h` e outras para lidar com funções relacionadas a alocação de memória, chamadas de sistema, threads e tempo.

```
1 #define _GNU_SOURCE
2 #include <stdlib.h>
3 #include <malloc.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #include <signal.h>
7 #include <sched.h>
8 #include <stdio.h>
9 #include <pthread.h> // Adicionamos a biblioteca pthread para usar mutex
10 #include <time.h>    // Adicionamos a biblioteca time para inicializar a semente aleatória
11
```

Nesta seção, um tamanho de pilha (stack) é definido para as threads (`FIBER_STACK`) e uma estrutura conta é definida para representar uma conta bancária com um saldo inteiro. Um array de duas contas (`contas[2]`) é declarado para simular duas contas bancárias. A variável `valor` será o valor da transferência.

```
12 // 64kB stack
13 #define FIBER_STACK (1024 * 64)
14
15 struct c {
16     int saldo;
17 };
18
19 typedef struct c conta;
20 conta contas[2]; // Alteramos para um array de contas
21 int valor;
22
```

O programa cria 100 threads que realizam transferências concorrentes entre duas contas bancárias simuladas. O mutex é usado para garantir a exclusão mútua e evitar condições de corrida. Após todas as threads terminarem, a memória alocada é liberada e o mutex é destruído. Isso atende aos objetivos de abordar a concorrência, consumir chamadas de sistema do SO e usar as chamadas de sistema de forma adequada.

```
57 int main()
58 {
59     void* stack;
60     pid_t pid;
61     int i;
62
63     // Inicializa o mutex
64     pthread_mutex_init(&mutex, NULL);
65
66     // Inicializa a semente aleatória
67     srand(time(NULL));
68
69     // Aloca o stack
70     stack = malloc(FIBER_STACK);
71     if (stack == 0) {
72         perror("malloc: could not allocate stack");
73         exit(1);
74     }
75
76     // Inicializa as contas com saldo 100
77     for (i = 0; i < 2; i++) {
78         contas[i].saldo = 100;
79     }
80
81     printf("Transferindo 10 entre contas aleatórias\n");
82     valor = 10;
83
84     // Cria 100 threads para realizar as transferências
85     pthread_t threads[100];
86     for (i = 0; i < 100; i++) {
87         if (pthread_create(&threads[i], NULL, transferencia, NULL) != 0) {
88             perror("pthread_create");
89             exit(2);
90         }
91     }
92
93     // Espera que todas as threads terminem
94     for (i = 0; i < 100; i++) {
95         pthread_join(threads[i], NULL);
96     }
97
98     // Libera o stack
99     free(stack);
100
101     // Destrói o mutex
102     pthread_mutex_destroy(&mutex);
103
104     printf("Transferências concluídas e memória liberada.\n");
105     return 0;
106 }
```

A transferência escolhe contas aleatórias, verifica se há saldo suficiente na conta de débito, realiza a transferência de dinheiro, atualiza os saldos das contas, imprime mensagens informativas e repete o processo 10 vezes. O uso do mutex garante que as operações de débito e crédito nas contas sejam feitas de forma segura e exclusiva por uma thread de cada vez.

```
26 // Função de transferência
27 void* transferencia(void* arg)
28 {
29     int i;
30     for (i = 0; i < 10; i++) {
31         // Escolhe contas aleatoriamente
32         int conta_debitar = rand() % 2;
33         int conta_creditar = 1 - conta_debitar;
34
35         pthread_mutex_lock(&mutex); // Bloqueia o acesso às contas
36         if (contas[conta_debitar].saldo >= valor) {
37             contas[conta_debitar].saldo -= valor;
38             contas[conta_creditar].saldo += valor;
39         }
40         pthread_mutex_unlock(&mutex); // Libera o acesso às contas
41
42         // Constrói as mensagens
43         char msg_saldo_conta0[100];
44         char msg_saldo_conta1[100];
45         char msg_transferencia[100];
46
47         snprintf(msg_saldo_conta0, sizeof(msg_saldo_conta0), "Saldo da conta 0: %d\n", contas[0].saldo);
48         snprintf(msg_saldo_conta1, sizeof(msg_saldo_conta1), "Saldo da conta 1: %d\n", contas[1].saldo);
49         snprintf(msg_transferencia, sizeof(msg_transferencia), "Transferência concluída com sucesso!\n");
50
51         // Imprime as mensagens
52         printf("%s%s%s", msg_saldo_conta0, msg_saldo_conta1, msg_transferencia);
53     }
54     return NULL;
55 }
```

## Fontes Bibliográficas

UNIVESP. Sistemas Operacionais – Aula 03 - Chamada de Sistema e Interrupção. 18 de maio de 2017. Disponível em:

<<https://www.youtube.com/watch?v=IE9EVxy8Ups&list=PLxI8Can9yAHeK7GUEGxMsqoPRmJKwI9Jw&index=4>>

UNIVESP. Sistemas Operacionais – Aula 07 – Threads. 22 de maio de 2017. Disponível em:

<<https://www.youtube.com/watch?v=Tbwu55lov5s&list=PLxI8Can9yAHeK7GUEGxMsqoPRmJKwI9Jw&index=8>>