Utilização de semáforo na solução de condição de corrida

A listagem do Anexo I mostra um programa com 2 threads que são executados concorrentemente. O thread "**funcaoDeposito**" realiza uma operação de depósito de 100 em uma conta, enquanto o thread "**funcaoSaque**" realiza um saque de 200 na mesma conta.

A função responsável por atualizar o saldo é a "**atualiza_saldo**". De modo a provocar a situação de haver dois threads dentro da região crítica, foi introduzida a chamada de sistema usleep(1000), que faz com que a função bloqueie por 1ms. Esse tempo será suficiente para provocar a ocorrência da condição de corrida no processo, em que os dois threads tentarão atualizar a mesma variável simultaneamente, fazendo com que o resultado seja inconsistente.

Durante a execução, os dois threads solicitarão a atualização do saldo em "**registro**". Como ambos leem a varável antes de alterá-la, somente uma das alterações terá efeito.

O resultado do processamento do programa do Anexo I será:

```
Saldo antes das operações = 500
Iniciando operação [-200]
Iniciando operação [100]
Terminada operação [-200]
Terminada operação [100]
Saldo depois das operações = 600
```

Você pode perceber que é iniciada a operação de saque de 200 e, antes que ela termine, é iniciada a operação de depósito de 100. O saldo antes das operações era 500 e o resultado do processamento deveria ser 400. Porém, devido à condição de corrida, o resultado do processamento foi 600.

Para a solução deste problema, é necessária a definição de uma região crítica e sua proteção por semáforo, de modo que possa haver somente um processo em execução dentro da região crítica.

Na listagem no Anexo II, você encontra o mesmo programa, com a inserção da região crítica protegida por um semáforo de nome **mutex** (linhas 43 a 50). O semáforo impede que dois threads executem concorrente dentro da região crítica, impedindo o acesso simultâneo ao saldo da conta.

O resultado do processamento do programa do Anexo II será:

```
Saldo antes das operações = 500
Iniciando operação [-200]
Terminada operação [-200]
Iniciando operação [100]
```

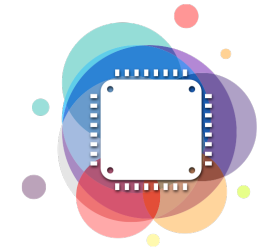

```
Terminada operação [100]
Saldo depois das operações = 400
```

Pela saída do processamento, é possível verificar que é iniciada a operação de saque de 200 e que a operação de depósito de 100 é iniciada somente após o término da operação de saque. Como o semáforo permite a execução de apenas um programa dentro da região crítica, não ocorre a condição de corrida.

ANEXO I

Programa sem semáforo, sujeito à condição de corrida.

```
 1  #define _GNU_SOURCE
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <unistd.h>
 5  #include <sched.h>
 6  #include <sys/wait.h>
 7
 8  #define TAMANHO_PILHA 65536
 9
10  typedef struct {
11    double saldo;
12  } Registro;
13
14  Registro registro;
15
16  Registro le_registro(int conta){
17    return registro;
18  }
19
20  void grava_registro(Registro reg, int conta){
21    registro = reg;
22  }
23
24  void atualiza_saldo(double valor, int conta) {
25    Registro registro;
26    printf("Iniciando operação [%.2f]\n", valor);
27    registro = le_registro(conta);
28    usleep(1000);
29    registro.saldo = registro.saldo + valor;
30    grava_registro(registro, conta);
31    printf("Terminada operação [%.2f]\n", valor);
32  }
33
34  int funcaoDeposito(void *arg) {
35    // Faz deposito de 100,00
36    atualiza_saldo(100, 0);
37  }
38
39  int funcaoSaque(void *arg) {
40    // Faz saque de 200,00
41    atualiza_saldo(-200, 0);
42  }
43
44  int main() {
45    void *pilha1, *pilha2;
```

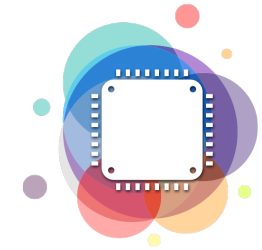



```
46    int pid1, pid2;
47
48    registro.saldo = 500; // Inicializa saldo
49    printf("Saldo antes das operações = %.2f\n", registro.saldo);
50
51    // Aloca pilha para thread de depósito
52    if ((pilha1 = malloc(TAMANHO_PILHA)) == 0) {
53      perror("Erro na alocação da pilha.");
54      exit(1);
55    }
56    // Inicia thread de depósito
57    pid1 = clone(funcaoDeposito,
58           pilha1 + TAMANHO_PILHA,
59           CLONE_VM | SIGCHLD,
60           NULL);
61
62    // Aloca pilha para thread de saque
63    if ((pilha2 = malloc(TAMANHO_PILHA)) == 0) {
64      perror("Erro na alocação da pilha.");
65      exit(1);
66    }
67    // Inicia thread de saque
68    pid2 = clone(funcaoSaque,
69           pilha2 + TAMANHO_PILHA,
70           CLONE_VM | SIGCHLD,
71           NULL);
72
73    //Aguarda final da operações
74    waitpid(pid1, 0, 0);
75    waitpid(pid2, 0, 0);
76
77    printf("Saldo depois das operações = %.2f\n", registro.saldo);
78  }
```

ANEXO II

Programa com semáforo, solucionando a condição de corrida.

```
 1  /*********************************\
 2  *                                 *
 3  * COMPILAR COM PARÂMETRO -pthread *
 4  *                                 *
 5  \*********************************/
 6
 7  #define _GNU_SOURCE
 8  #include <stdio.h>
 9  #include <stdlib.h>
10  #include <unistd.h>
11  #include <sched.h>
12  #include <sys/wait.h>
13  #include <semaphore.h>
14
15  #define TAMANHO_PILHA 65536
16
17  sem_t mutex; // Cria semáforo mutex
18
19  void up(sem_t *sem) {
20    sem_wait(sem);
21  }
22
23  void down(sem_t *sem) {
24    sem_post(sem);
25  }
26
27  typedef struct {
28    double saldo;
29  } Registro;
30
31  Registro registro;
32
33  Registro le_registro(int conta){
34    return registro;
35  }
36
37  void grava_registro(Registro reg, int conta){
38    registro = reg;
39  }
40
41  void atualiza_saldo(double valor, int conta) {
42    Registro registro;
43    up(&mutex);
44    printf("Iniciando operação [%.2f]\n", valor);
45    registro = le_registro(conta);
```




```
46    usleep(1000);
47    registro.saldo = registro.saldo + valor;
48    grava_registro(registro, conta);
49    printf("Terminada operação [%.2f]\n", valor);
50    down(&mutex);
51  }
52
53  int funcaoDeposito(void *arg) {
54    // Faz deposito de 100,00
55    atualiza_saldo(100, 0);
56  }
57
58  int funcaoSaque(void *arg) {
59    // Faz saque de 200,00
60    atualiza_saldo(-200, 0);
61  }
62
63  int main() {
64    void *pilha1, *pilha2;
65    int pid1, pid2;
66
67    // Inicializa mutex om valor 1 (somente um thread na região crítica)
68    sem_init(&mutex, 1, 1);
69
70    registro.saldo = 500; // Inicializa saldo
71    printf("Saldo antes das operações = %.2f\n", registro.saldo);
72
73    // Aloca pilha para thread de depósito
74    if ((pilha1 = malloc(TAMANHO_PILHA)) == 0) {
75      perror("Erro na alocação da pilha.");
76      exit(1);
77    }
78    // Inicia thread de depósito
79    pid1 = clone(funcaoDeposito,
80           pilha1 + TAMANHO_PILHA,
81           CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD,
82           NULL);
83
84    // Aloca pilha para thread de saque
85    if ((pilha2 = malloc(TAMANHO_PILHA)) == 0) {
86      perror("Erro na alocação da pilha.");
87      exit(1);
88    }
89    // Inicia thread de saque
90    pid2 = clone(funcaoSaque,
91           pilha2 + TAMANHO_PILHA,
92           CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD,
93           NULL);
94
95    //Aguarda final da operações
```


```
 96    waitpid(pid1, 0, 0);
 97    waitpid(pid2, 0, 0);
 98
 99    printf("Saldo depois das operações = %.2f\n", registro.saldo);
100  }
```