Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programação Concorrente

Trabalho Extraclasse

Aluno(s): Henrique Mendes de Freitas Mariano; Leonardo Rodrigues de Souza
Matrícula: 170012280; 170060543

1 - Problema dos macacos

1.1 - Descrição do problema

Nos dias atuais a natureza e a tecnologia são coisas que convivem conjuntamente e assim podem acontecer interações entre as duas. Logo, suponha que haja macacos em ambos os lados da margem de um rio, e que frequentemente os macacos decidem passar de um lado para o outro à procura de comida. O caminho de um lado para o outro é feito através de uma ponte estreita. Mais de um macaco pode atravessar a ponte ao mesmo tempo, mas isso só é possível se eles estiverem indo na mesma direção.
Como na natureza há diversas espécies de animais, temos uma outra espécie de macaco que atravessa a ponte, o gorila. Dado que os gorilas são muito pesados, eles só podem atravessar a ponte sozinhos. Como os outros macacos têm medo dos gorilas, eles têm preferência para fazer a travessia.

1.2 - Solução da primeira parte do problema (macacos sem gorilas)

Para solucionar este problema, foi feita uma representação de cada macaco como uma struct que possui uma variável side que indica o lado que o macaco está e uma thread associada a esse macaco. Além disso, um mutex que garante o turno para cada thread também foi empregado, para garantir a exclusividade de um lado da ponte por vez. E por fim, o dois mutex's, um para cada lado, garante a exclusividade de cada thread as variáveis de contagem de macacos de cada lado. Para o bom funcionamento do programa, inicialmente quando cada macaco tenta entrar na ponte, ele chama o mutex de turno, e garante sua exclusividade em relação a todos os outros macacos. Em seguida, ele chama o mutex do seu lado da ponte, assim poderá alterar as variáveis de contagem sem problemas.

Por fim, ele libera o mutex do seu lado da ponte, permitindo que outros macacos do mesmo lado também atravessem a ponte. E por fim, ele libera o mutex de turno, permitindo que o outro lado da ponte inicie sua travessia. Ao final, o macaco altera sua struct e com seu novo lado, e como cada macaco possui uma struct em um lugar diferente da memória, não é necessário controle de escrita aqui, já que não ocorre uma condição de corrida.

```c
// autores: Henrique Mendes de Freitas Mariano e Leonardo Rodrigues de Souza
// arquivo: macaco.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

typedef enum {
    LEFT, RIGHT
} side;

pthread_mutex_t leftMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rightMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t entrada = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t turno = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t print = PTHREAD_MUTEX_INITIALIZER;

typedef struct t_monkey {
    side side;
    pthread_t thread;
} monkey;

monkey macacos[1001];

int countLeftSide = 0, countRightSide = 0, totalLeftSide = 0, totalRightSide
= 0;
```

```
void *runMonkey(void* id) {
    int i = *((int*) id);
    while(1){
        monkey m = macacos[i];
        pthread_mutex_t* sideMutex = ((m.side == LEFT)? &leftMutex :
&rightMutex);
        pthread_mutex_lock(&turno);
        pthread_mutex_lock(sideMutex);
        if(m.side == LEFT) totalLeftSide--, totalRightSide++,
countLeftSide++;
        else totalLeftSide++,totalRightSide--, countRightSide++;

        if ((m.side == LEFT && countLeftSide == 1) || (m.side == RIGHT &&
countRightSide == 1)) pthread_mutex_lock(&entrada);
        pthread_mutex_unlock(sideMutex);
        pthread_mutex_unlock(&turno);
        pthread_mutex_lock(&print);
        printf ("[%d/%d] ", totalLeftSide, totalRightSide);
        if(m.side == LEFT) printf("Macaco %d passando da esquerda para
direita\n", i);
        else printf("Macaco %d passando da direita para esquerda\n", i);
        pthread_mutex_unlock(&print);
        macacos[i].side = (m.side == LEFT)? RIGHT : LEFT;
        sleep(1);

        pthread_mutex_lock(sideMutex);
        if(m.side == LEFT) countLeftSide--;
        else countRightSide--;
        if ((m.side == LEFT && countLeftSide == 0) || (m.side == RIGHT &&
countRightSide == 0)) pthread_mutex_unlock(&entrada);
        pthread_mutex_unlock(sideMutex);
    }
}

int main(void){
    int n = 0;
    int *id = NULL;
```

```
    printf("Número de macacos em cada lado (1 a 1000): ");
    do {
        scanf("%d", &n);
    } while (n <= 0 || n >= 1000);

    pthread_mutex_init(&leftMutex, NULL);
    pthread_mutex_init(&rightMutex, NULL);
    pthread_mutex_init(&entrada, NULL);
    pthread_mutex_init(&turno, NULL);
    pthread_mutex_init(&print, NULL);
    totalLeftSide = n, totalRightSide = n;


    for(int i = 0; i < 2 * n; i++){
        id = (int *) calloc(1, sizeof(int));
        *id = i;
        macacos[(*id)].side = (i % 2) == 0? LEFT : RIGHT;
        pthread_create(&macacos[(*id)].thread, NULL, runMonkey, (void *)
id);
    }

    for(int i = 0; i < 2 * n; i++) pthread_join(macacos[i].thread, NULL);

    return 0;
}
```

1.2 - Solução da segunda parte do problema (macacos com gorilas)

Já na segunda parte do problema é necessário garantir a prioridade dos gorilas. Quando um gorila for atravessar a ponte, o mesmo deve ser o único a fazer a travessia. Portanto, outros gorilas e macacos devem esperar até que este atravesse. Para isto, o uso de mutex e sinais não seria muito úteis, isso porque deve-se garantir o bom funcionamento de todas as threads, e somente quando o gorila for passar que este deve ter exclusividade. E por isso, o uso de uma espera ocupada, nas threads dos macacos, foi empregada. Assim, ao iniciar uma travessia, a thread do gorila seta uma variável que irá bloquear as threads dos macacos. E por

isso, o gorila demora 3 segundos para fazer a travessia, e durante todo esse tempo nenhum gorila ou macaco pode passar, e ao fim da travessia o gorila dorme por mais 5 segundos.

```c
// autores: Henrique Mendes de Freitas Mariano e Leonardo Rodrigues de Souza
// arquivo: gorila.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define RED   "\x1B[31m"
#define RESET "\x1B[0m"

typedef enum {
    LEFT, RIGHT
} side;

pthread_mutex_t leftMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rightMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t entrada = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t turno = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t print = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t gorila = PTHREAD_MUTEX_INITIALIZER;

typedef struct t_monkey {
    side side;
    pthread_t thread;
} monkey;

monkey macacos[1001], gorilas[1001];

int countLeftSide = 0, countRightSide = 0, totalLeftSide = 0, totalRightSide
= 0;
int gorilaOcupado;
int gorila_passando = 0;
```

```
void *runGorila(void* id) {
    int i = *((int*) id);
    while(1){
        monkey g = gorilas[i];
        pthread_mutex_t* sideMutex = ((g.side == LEFT)? &leftMutex :
&rightMutex);
        pthread_mutex_lock(&turno);
        pthread_mutex_lock(sideMutex);
        gorilaOcupado = 1;
        if(g.side == LEFT) totalLeftSide--, totalRightSide++,
countLeftSide++;
        else totalLeftSide++,totalRightSide--, countRightSide++;

        if ((g.side == LEFT && countLeftSide == 1) || (g.side == RIGHT &&
countRightSide == 1)) pthread_mutex_lock(&entrada);

        pthread_mutex_unlock(sideMutex);
        pthread_mutex_unlock(&turno);
        pthread_mutex_lock(&print);
        printf ("[%d/%d] ", totalLeftSide, totalRightSide);
        if(g.side == LEFT) printf(RED "Gorila %d passando da esquerda para
direita...\n" RESET, i);
        else printf(RED "Gorila %d passando da direita para esquerda....\n"
RESET, i);
        sleep(3);
        printf ("[%d/%d] ", totalLeftSide, totalRightSide);
        if(g.side == LEFT) printf(RED "Gorila %d passou da esquerda para
direita!\n" RESET, i);
        else printf(RED "Gorila %d passando da passou para esquerda!\n"
RESET, i);
        pthread_mutex_unlock(&print);
        gorilas[i].side = (g.side == LEFT)? RIGHT : LEFT;
        sleep(1);
        pthread_mutex_lock(sideMutex);
        if(g.side == LEFT) countLeftSide--;
        else countRightSide--;
```

```
        if ((g.side == LEFT && countLeftSide == 0) || (g.side == RIGHT &&
countRightSide == 0)) pthread_mutex_unlock(&entrada);
        gorilaOcupado = 0;
        pthread_mutex_unlock(sideMutex);
        sleep(5);
    }
}

void *runMonkey(void* id) {
    int i = *((int*) id);
    while(1){
        monkey m = macacos[i];
        pthread_mutex_t* sideMutex = ((m.side == LEFT)? &leftMutex :
&rightMutex);
        do { } while (gorilaOcupado == 1);
        pthread_mutex_lock(&turno);
        pthread_mutex_lock(sideMutex);
        if(m.side == LEFT) totalLeftSide--, totalRightSide++,
countLeftSide++;
        else totalLeftSide++,totalRightSide--, countRightSide++;

        if ((m.side == LEFT && countLeftSide == 1) || (m.side == RIGHT &&
countRightSide == 1)) pthread_mutex_lock(&entrada);
        pthread_mutex_unlock(sideMutex);
        pthread_mutex_unlock(&turno);
        pthread_mutex_lock(&print);
        printf ("[%d/%d] ", totalLeftSide, totalRightSide);
        if(m.side == LEFT) printf("Macaco %d passando da esquerda para
direita\n", i);
        else printf("Macaco %d passando da direita para esquerda\n", i);
        pthread_mutex_unlock(&print);
        macacos[i].side = (m.side == LEFT)? RIGHT : LEFT;
        sleep(1);
        pthread_mutex_lock(sideMutex);
        if(m.side == LEFT) countLeftSide--;
        else countRightSide--;
```

```
        if ((m.side == LEFT && countLeftSide == 0) || (m.side == RIGHT &&
countRightSide == 0)) pthread_mutex_unlock(&entrada);
        pthread_mutex_unlock(sideMutex);
    }
}

int main(void){
    int n = 0;
    int n_gorilas = 0;
    int *id = NULL;
    int *id_gorila = NULL;
    gorilaOcupado = 0;
    printf("Número de macacos em cada lado (1 a 500): ");
    do {
        scanf("%d", &n);
    } while (n <= 0 || n >= 1000);

    printf("Número de gorilas em cada lado (1 a 500): ");
    do {
        scanf("%d", &n_gorilas);
    } while (n_gorilas <= 0 || n_gorilas >= 1000);

    pthread_mutex_init(&leftMutex, NULL);
    pthread_mutex_init(&rightMutex, NULL);
    pthread_mutex_init(&entrada, NULL);
    pthread_mutex_init(&turno, NULL);
    pthread_mutex_init(&print, NULL);
    totalLeftSide = n + n_gorilas, totalRightSide = n + n_gorilas;

    for(int i = 0; i < 2 * n; i++){
        id = (int *) calloc(1, sizeof(int));
        *id = i;
        macacos[(*id)].side = (i % 2) == 0? LEFT : RIGHT;
        pthread_create(&macacos[(*id)].thread, NULL, runMonkey, (void *)
id);
    }
```

```
    for (int i = 0; i < 2 * n_gorilas; i++){
        id_gorila = (int *) calloc(1, sizeof(int));
        *id_gorila = i;
        gorilas[(*id_gorila)].side = (i % 2) == 0? LEFT : RIGHT;
        pthread_create(&gorilas[(*id_gorila)].thread, NULL, runGorila, (void
*) id_gorila);
    }

    for(int i = 0; i < 2 * n; i++) pthread_join(macacos[i].thread, NULL);
    for(int i = 0; i < 2 * n_gorilas; i++) pthread_join(gorilas[i].thread,
NULL);

    return 0;
}
```

Nas duas soluções foi utilizado um lock para realizar os prints.