Questionário - 12

Arthur C. M. Barcella e Matheus P. Salazar

Usando semáforos, escreva o pseudocódigo de um sistema produtor/consumidor com dois buffers limitados organizado na forma X => B1 => Y => B2 => Z, onde X, Y e Z são tipos de processos e B1 e B2 são buffers independentes com capacidades N1 e N2, respectivamente, inicialmente vazios. Os buffers são acessados unicamente através das operações insere(Bi , item) e retira(Bi , item) (que não precisam ser detalhadas). O número de processos X, Y e Z é desconhecido. Devem ser definidos os códigos dos processos X, Y e Z e os semáforos necessários, com seus significados e valores iniciais.

```
semaforo Buffer1_item = 0 // inicialmente vazio
semaforo Buffer1_item = 0 // inicialmente vazio
semaforo Buffer1_vagas = N1 // inicialmente todas as vagas livres
semaforo Buffer2_vagas = N2 // inicialmente todas as vagas livres
semaforo mutex_Buffer1 = 1 // inicialmente livre
semaforo mutex_Buffer2 = 1 // inicialmente livre

ProdutorX {
    while (true) {
        item_X = produz_X()
        down(Buffer1_vagas)
        down(mutex_Buffer1)
        insere(Buffer1, item_X)
        up(mutex_Buffer1)
        up(Buffer1_item)
    }
}

ConsumidorY {
    while (true) {
        down(Buffer1_item)
        down(mutex_Buffer1)
        item_X = retira(Buffer1)
        up(mutex_Buffer1)
        up(Buffer1_vagas)
        consome(item_X)
    }
}

```

```
ProdutorY {
    while (true) {S
        item_Y = produz_Y()
        down(Buffer2_vagas)
        down(mutex_Buffer2)
        insere(Buffer2, item_Y)
        up(mutex_Buffer2)
        up(Buffer1_item)
    }
}

ConsumidorZ {
    while (true) {
        down(Buffer1_item)
        down(mutex_Buffer2)
        item_Y = retira(Buffer2)
        up(mutex_Buffer2)
        up(Buffer2_vagas)
        consome(item_Y)
    }
}
```

O trecho de código a seguir apresenta uma solução para o problema do jantar dos filósofos, mas ele contém um erro. Explique o código e explique onde está o erro e porque ele ocorre. A seguir, modifique o código para que ele funcione corretamente.

```
#define N 5
sem_t garfo[N] ; // 5 semáforos iniciados em 1
void filosofo (int i) {
        while (1) {
                medita () ;
                sem_down (garfo [i]) ;
                sem_down (garfo [(i+1) % N]) ;
                come () ;
                sem_up (garfo [i]) ;
                sem_up (garfo [(i+1) % N]) ;
        }
}
```

O trecho de código a seguir apresenta uma solução para o problema do jantar dos filósofos, mas ele contém um erro. Explique o código e explique onde está o erro e porque ele ocorre. A seguir, modifique o código para que ele funcione corretamente.

```c
#define N 5

semaforo garfo[N];

void filosofo (int i) {
    while (1) {
        medita ();
        sem_down (garfo [i]);
        sem_down (garfo [(i+1) % N]);
        come ();
        sem_up (garfo [i]);
        sem_up (garfo [(i+1) % N]);
    }
}

```

```c
#define N 5

semaforo garfo[N];
semaforo saleiro;

void filosofo (int i) {
    while (1) {
        medita();
        sem_down(saleiro);
        sem_down(garfo[i]);
        sem_down(garfo[(i+1) % N]);
        sem_up(saleiro);
        come();
        sem_up(garfo[i]);
        sem_up(garfo[(i+1) % N]);
    }
}
```

O código mostrado acima representa um problema clássico de acesso a múltiplos recursos, onde cada consumidor tem acesso a dois recursos, no caso em questão são cinco filósofos e cinco garfos.

A função void filósofo recebe um argumento que representa o filósofo que está sendo tratado no momento pela thread ou processo, onde o mesmo executa continuamente e tem duas funções, meditar e comer, para comer tem que ter em mãos dois recursos de garfo, para fazer o controle de acesso ao recurso temos dois semáforos que representam cada garfo, pois os garfos tem concorrência, logo precisa de um semáforo controlando o acesso ao recurso.

O erro do código é que pode haver um impasse com os garfos, pois se cada um dos filósofos pegar o garfo à sua direita acaba que todos terão o acesso ao recurso bloqueado, pois todos pegaram seus “garfos”, logo cada filósofo “a sua frente" está com um recurso bloqueado e dessa forma não conseguem ter os dois recursos necessários para entrar em estado de processamento, gerando um impasse.

Suponha três robôs (Bart, Lisa, Maggie), cada um controlado por sua própria thread. Você deve escrever o código das threads de controle, usando semáforos para garantir que os robôs se movam sempre na sequência Bart => Lisa => Maggie => Lisa => Bart => Lisa => Maggie => · · ·, um robô de cada vez. Use a chamada move() para indicar um movimento do robô. Não esqueça de definir os valores iniciais das variáveis e/ou dos semáforos utilizados. Soluções envolvendo espera ocupada (busy wait) não devem ser usadas.

```c
// Bibliotecas

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

// Variáveis globais

sem_t semaforoLisa;
sem_t semaforoBart;
sem_t semaforoMaggie;
int escalonado = 0;

void *move_bart(void *arg) {
    while(1) {
        sem_wait(&semaforoBart);
        printf("movimento Bart\n");
        sem_post(&semaforoLisa);
    }
}
```

```c
void *move_lisa(void *arg) {
    while(1) {
        sem_wait(&semaforoLisa);
        printf("movimento Lisa\n");
        if(escalonado == 0){
            sem_post(&semaforoMaggie);
            escalonado = 1;
        }
        else{
            sem_post(&semaforoBart);
            escalonado = 0;
        }
    }
}

void *move_maggie(void *arg) {
    while(1) {
        sem_wait(&semaforoMaggie);
        printf("movimento Maggie\n");
        sem_post(&semaforoLisa);
    }
}
```

```c
int main (int argc, char *argv[]) {

    sem_init(&semaforoLisa, 0, 0);
    sem_init(&semaforoBart, 0, 1);
    sem_init(&semaforoMaggie, 0, 0);

    pthread_t thread_bart;
    pthread_t thread_lisa;
    pthread_t thread_maggie;

    pthread_create(&thread_bart, NULL, move_bart, NULL);
    pthread_create(&thread_lisa, NULL, move_lisa, NULL);
    pthread_create(&thread_maggie, NULL, move_maggie, NULL);

    pthread_join(thread_bart, NULL);
    pthread_join(thread_lisa, NULL);
    pthread_join(thread_maggie, NULL);

    return 0;
}
```

O Rendez-Vous é um operador de sincronização forte entre dois processos ou threads, no qual um deles espera até que ambos cheguem ao ponto de encontro (rendez-vous, em francês). O exemplo a seguir ilustra seu uso:

```
Processo A                    Processo B

    A1 () ;                       B1 () ;

    rv_wait (rv) ;                rv_wait (rv) ;

    A2 () ;                       B2 () ;

    rv_wait (rv) ;                rv_wait (rv) ;

    A3 () ;                       B3 () ;
```

```c
// bibliotecas
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

// semáforos para sincronização:
sem_t sem1, sem2;

// função executada pela thread 1
void* thread1(void* arg) {

    // imprime mensagem de chegada a ponto de sincronização
    printf("Thread 1 chegou ao ponto de sincronização (Rendez-Vous).\n");
    //-------------------------------------------------------------

    // semáforo 1 incrementa (libera semáforo 1): Ao liberar o semafaro 1,
    // a thread 2 que estava aguardando o semáforo 1 é liberada.
    sem_post(&sem1);

    // semáforo 2 aguarda (bloqueia semáforo 2): Ao bloquear o semáforo 2,
    // a thread 1 aguarda a liberação do semáforo 2 pela thread 2.
    sem_wait(&sem2);

    //-------------------------------------------------------------
    // imprime mensagem de saída do ponto de sincronização
    printf("Thread 1 saiu do ponto de sincronização (Rendez-Vous).\n");
    pthread_exit(NULL);
}

```

Nota: O código da Thread 1 e 2 são iguais, dessa forma, não adicionei a apresentação. Quando uma das Thread chega ao ponto de sincronização ela incrementa o semáforo da outra tarefa, e aguarda seu próprio semáforo, permitindo assim que as duas tarefas "rodem" ao mesmo tempo assim que a outra tarefa chegar no ponto de sincronização e liberar a tarefa inicial.

```c
int main() {

  // declaração das threads
    pthread_t process1, process2;

    // inicializa semáforos de sincronização

    // semáforo 1 é inicializado com valor 0, pois a thread 1
    // deve aguardar a thread 2 para liberar o semáforo 1.
    sem_init(&sem1, 0, 0);

    // semáforo 2 é inicializado com valor 0, pois a thread 2
    // deve aguardar a thread 1 para liberar o semáforo 2.
    sem_init(&sem2, 0, 0);

    // cria as threads.
    // parâmetros:
      // 1. Ponteiro para a thread (process1)
      // 2. Atributos da thread (NULL = padrão)
      // 3. função que a thread deve executar (função thread1)
      // 4. parâmetro da função que a thread deve executar (sem parâmetro)
    pthread_create(&process1, NULL, thread1, NULL);
    pthread_create(&process2, NULL, thread2, NULL);

    // espera as threads terminarem
    pthread_join(process1, NULL);
    pthread_join(process2, NULL);

    return 0;
}
```

Uma Barreira é um operador de sincronização forte entre N processos ou threads, no qual eles esperam até que todos cheguem à barreira. O exemplo a seguir ilustra seu uso:

```
Processo A
    A1 () ;
    barrier_wait (b) ;
    A2 () ;
    barrier_wait (b) ;
    A3 () ;
Processo B
    B1 () ;
    barrier_wait (b) ;
    B2 () ;
    barrier_wait (b) ;
    B3 () ;
```

```
Processo C
    C1 () ;
    barrier_wait (b) ;
    C2 () ;
    barrier_wait (b) ;
    C3 () ;
Processo D
    D1 () ;
    barrier_wait (b) ;
    D2 () ;
    barrier_wait (b) ;
    D3 () ;
```

No início da função "processo", o ID do processo é impresso na tela. Após isso, é feita uma exclusão mútua usando o semáforo "mutex" para garantir que apenas um processo por vez incrementa a variável "count", que conta quantos processos chegaram à barreira.

Se o número de processos que chegaram à barreira for igual a N, todos os processos são liberados usando o semáforo "barreira". Se não, o processo continua esperando na barreira usando a função "sem_wait(&barreira)".

```c
// Bibliotecas:
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

// número de processos
#define N 5

// semáforo para exclusão mútua
sem_t mutex;

// semáforo para sincronização
sem_t barreira;

// contador de processos que chegaram à barreira
int count = 0;
```

```c
// função executada por cada processo
void *processo(void *identificador) {

    // obtém o identificador de cada processo.
    int id = *(int*)identificador;

    // imprime o identificador do processo
    printf("Processo %d chegou à barreira.\n", id);

    // Início da exclusão mútua
    sem_wait(&mutex);

    // Espera todos os processos chegarem à barreira
    // para liberar todos os processos.
    count++;
    if (count == N) {

        // libera todos os processos que estão esperando
        // Quando o último processo chegar à barreira.
        sem_post(&barreira);
    }

    // fim da exclusão mútua
    sem_post(&mutex);

    // espera pelo último processo
    // para liberar os outros processos
    sem_wait(&barreira);
    sem_post(&barreira);

    // imprime o identificador do processo, após liberar
    // todos os processos
    printf("Processo %d passou pela barreira.\n", id);

    // termina a thread
    pthread_exit(NULL);
}
```

```c
int main() {

    // vetor de threads, sendo que N é o valor de processos a serem criados
    pthread_t threads[N];

    // vetor de identificadores
    int ids[N];

    // variável auxiliar
    int i;

    // inicializa os semáforos

    // mutex = 1, pois o primeiro processo que chegar à barreira deve ser liberado.
    sem_init(&mutex, 0, 1);

    // barreira = 0, pois nenhum processo deve ser liberado.
    // O último processo que chegar à barreira deve liberar todos os processos.
    sem_init(&barreira, 0, 0);

    // cria as threads
    // cada thread executa a função processo
    // e recebe como parâmetro o identificador do processo
    for (i = 0; i < N; i++) {
```

```c
    for (i = 0; i < N; i++) {

        // atribui o identificador do processo ao vetor de identificadores
        ids[i] = i;

        // cria a thread

        // Parâmetros na criação da thread:
        // 1. Ponteiro para a thread (threads[i])
        // 2. Atributos da thread (NULL = padrão)
        // 3. função que a thread deve executar (função processo)
        // 4. parâmetro da função que a thread deve executar (identificador do processo)

        if (pthread_create(&threads[i], NULL, processo, &ids[i])) {
            printf("Erro ao criar a thread %d.\n", i);
            exit(EXIT_FAILURE);
        }
    }

    // espera as threads terminarem:
    for (i = 0; i < N; i++) {
        if (pthread_join(threads[i], NULL)) {
            printf("Erro ao esperar pela thread %d.\n", i);
            exit(EXIT_FAILURE);
        }
    }
    return 0;
}
```

Questionário - 12

Arthur C. M. Barcella e Matheus P. Salazar