Programação Concorrente - Trabalho 2022/2

Gabriel da Silva Corvino Nogueira - 18/0113330

Janeiro de 2023

1 Introdução

No que tange ao assunto programas de computador, é comum que parte dos indivíduos com algum conhecimento sobre programação de computadores tenda a trata-lo como um processo similar a uma receita ou uma sequência de instruções a serem seguidas de forma sequencial ao longo das linhas que o descrevem [5]. Tal processo é chamado de programação sequencial, mas não é a única forma sob a qual um programa pode ser executado. Na realidade, existem programas capazes de definir ações que podem ocorrer ao mesmo tempo, denominados **programas concorrentes** [4].

De forma mais ampla, programação concorrente é o nome dado às notações e técnicas utilizadas para expressar potencial paralelismo e para resolver os problemas de sincronização e comunicação resultantes [3]. Além disso, programas concorrentes podem ser executados em vários processadores, de forma paralela, ou apenas em um processador, processo esse denominado pseudo-paralelismo. De qualquer forma, programas concorrentes realizam mais de um fluxo de execução ao mesmo tempo e compartilham recursos como memória e processamento. Como consequência, esses programas são mais difíceis de depurar, além de estarem suscetíveis às denominadas **condições de corrida**.

Condição de corrida é um erro que ocorre quando dois ou mais fluxos de execução acessam dados compartilhados e tentam altera-los ao mesmo tempo [6]. Tal processo torna os resultados de um programa imprevisíveis, visto que o resultado da execução vai depender da ordem em que o acesso à memória é feito. Dessa forma, os trechos do programa em que ocorre esse acesso compartilhado à memória compartilhada são chamados de **regiões críticas**.

Durante a disciplina, foram abordados diversos problemas clássicos da programação concorrente, juntamente com as técnicas utilizadas para sua resolução. Dentre as técnicas apresentadas, estão o uso de *locks*, variáveis de condição e semáforos. Isto posto, este trabalho tem o objetivo de desenvolver um algoritmo para tratar de problemas de comunicação entre processos através de memória compartilhado, por meio da utilização dos recursos ministrados em sala. Para a resolução do problema proposto, será feito o uso de duas dessas técnicas, sendo elas *locks* e variáveis de condição.

Um *lock* é um abstração que permite que no máximo uma thread tenha posse dele ao mesmo tempo [1]. Ele é utilizado para sincronizar processos e possui duas operações: **adquirir** e **liberar**. A operação de adquirir faz com que uma thread tome posse do *lock*. Dessa forma, se uma thread tenta adquirir um *lock* que está em posse de outra thread, ela será bloqueada até que o *lock* seja liberado. Por outro lado, variáveis de condição são mecanismos que permitem threads esperar a ocorrência de algum evento ou condição [2]. Tais mecanismos são dotados da operação *wait*, responsável por fazer uma thread esperar; da operação *signal*, responsável por acordar apenas uma thread, ou seja, fazer com que ela pare de esperar; e a operação *broadcast*, que é responsável por acordar todas a threads que se encontram no processo de espera.

2 Formalização do Problema Proposto

O problema proposto consiste no seguinte cenário: Existem duas ilhas produtoras de frutas (A e B), sendo a ilha A uma grande produtora da **abacates** e a ilha B uma grande produtora de **bananas**. Todavia, os habitantes da ilha A já estão cansados de comer abacates, ao passo que os habitantes da ilha B não aguentam mais comer bananas. Sendo assim, as ilhas fizeram um acordo no qual efetuariam a troca de bananas por abacates. Para tanto, seria utilizado um barco enviado pelo povo da ilha B, que estaria atracado nas prais da ilha A, contendo uma quantidade n caixas de bananas, sendo essa a capacidade máxima do navio (tanto para bananas quanto para abacates).

Dessa forma, cada habitante da ilha A deveria colher abacates das plantações da ilha e se dirigir ao navio para que possa trocar uma caixa de abacates por uma caixa de bananas. Uma vez que todos as n caixas de bananas tenham sido trocadas por caixas de abacates, o navio deve retornar para a ilha B, onde a troca continuará a ocorrer até que todas as caixas de abacates tenham sido trocadas por caixas de bananas e o navio possa zarpar novamente para a ilha A, reiniciando o ciclo.

Contudo, a população de cada ilha também é composta por alguns anciões, que, por estarem a comer bananas ou abacates por mais tempo, devem ter prioridade na hora de realizar a troca no navio. Dessa forma, toda vez que um ancião coletar a sua caixa e for em direção ao navio, ele deve ser capaz realizar a troca antes dos demais habitantes que estejam com uma caixa em mãos. Todavia, caso mais de um ancião esteja rumo a realizar a troca, eles devem competir entre si igualmente .

Isto posto, deve ser implementada uma solução concorrente para o problema, de forma que a troca entre as ilhas seja realizada e a prioridade dos anciões perante o restante dos habitantes da ilha seja mantida.

3 Descrição do Algoritmo Desenvolvido para Solução do Problema

A solução proposta para o problema assume que exitem $a_a + h_a$habitantes na ilha A e $a_b + h_b$ habitantes na ilha B, onde a_i corresponde ao número de anciões presentes na ilha i e h_i corresponde ao restante dos habitantes da ilha.

Em seguida, foram definidas quatro variáveis de condição, sendo elas: `zarpar_A`, responsável por controlar a saída do navio da ilha A; `habitantes_A`, responsável por liberar ou bloquear o processo de troca dos habitantes da ilha A; `zarpar_B` responsável por controlar a saída do navio da ilha A; e `habitantes_B`, responsável por liberar ou bloquear o processo de troca dos habitantes da ilha B.

Em sequência, é iniciado o processo de criação de threads. Primeiramente, são criadas h_a threads para representar os habitantes da ilha A e h_b threads para representar os habitantes da ilha B. Logo após, são criadas a_a threads

para simbolizar os anciões da ilha A e a_b threads para simbolizar os anciões da ilha B. Por fim, é criada uma única thread, que denota o navio.

Inicialmente, a função executada pela thread `navio` aguarda a variável de condição `zarpar_A` receber uma sinalização, o que corresponde ao navio atracado em A aguardando para que todas as caixas de bananas sejam trocadas por caixas de abacates. Após a thread ser liberada com a sinalização da variável de condição `zarpar_A`, a variável `navio_em_a` deve ser atribuída ao valor 0, para simbolizar que o navio deixou a ilha A. Em seguida, quando o navio chega à ilha B a variável `navio_em_b` deve ser atributa ao valor 1, representando que o navio se encontra na ilha B. Além disso, todas as threads bloqueadas por meio da variável de condição `habitantes_B` devem ser liberadas, o que indica que os habitantes da ilha B podem iniciar o processo de troca.

Após o processo de liberação descrito, a thread `navio` deve ser bloqueada e aguardar seu desbloqueio por meio da variável `zarpar_B`, o que indica que o navio está a esperar a realização do processo de trocas na ilha B. Prontamente, quando o navio puder zarpar, ele deve trocar imediatamente o valor da variável `navio_em_b` para 0 e, ao chegar à ilha B, alterar o valor de `navio_em_a` para 1, além de liberar as threads bloqueadas pela variável de condição `habitantes_A`, para que o processo de troca em A possa ser iniciado novamente, repetindo o ciclo inciado no parágrafo anterior.

Função executada pela thread `navio`.

```
void * f_navio(void *arg){

  printf("Navio: atracado em em A\n");
  while(1){

    pthread_mutex_lock(&mutex);

    // Navio aguarda ficar cheio em A
    pthread_cond_wait(&zarpar_A, &mutex);

    // Navio parte para B
    printf("Navio: partindo (sentido A->B)\n");
    navio_em_a = 0;
    sleep(10);

    // Navio chega em B
    printf("Navio: atracado em B\n");
    navio_em_b = 1;
    sleep(2);

    // Habitantes de B s o liberados para abastecer
    // o navio
    pthread_cond_broadcast(&habitantes_B);
```

```
    // Navio aguarda todas as trocas serem feitas
    // para zarpar
    pthread_cond_wait(&zarpar_B, &mutex);

    // Navio parte para A
    printf("Navio: partindo (sentido B->A)\n");
    navio_em_b = 0;
    sleep(10);

    // Navio chega em A
    printf("Navio: atracado em A\n");
    navio_em_a = 1;
    sleep(2);

    // Habitantes de A s o liberados para
    // abastecer o navio
    pthread_cond_broadcast(&habitantes_A);

    pthread_mutex_unlock(&mutex);
  }
}
```

No caso das threads que representam os habitantes da ilha A, é primeiramente realizado processo de colheita. Em seguida, o processo entra em uma região crítica, onde a thread deve verificar se o navio já atingiu sua capacidade de carga, se ele não está atracado na ilha e se algum ancião deseja fazer a troca de caixas. Caso alguma dessas condições seja verdadeira, a thread deve ser bloqueada por meio da variável de condição `habitantes_A` e aguardar seu desbloqueio. Uma vez desbloqueado, o habitante pode realizar a troca de caixas e, em seguida, deve verificar se após realizar sua troca não existem mais caixas de bananas no navio. Caso isso ocorra, deve ser enviado um sinal para a variável de condição `zarpar_A`, para que a thread que representa o navio possa ser desbloqueada, indicando que o processo de trocas foi finalizado. Finalmente, o habitante retorna a realizar a colheita e o processo de repete.

Função executada pelas threads que representam os habitantes de A.

```
void * f_habitanteA(void *arg){

  int id = *((int*)arg);
  while(1){

    // Habitante colhe abacates
    printf("Habitante %d (ilha A): \
    colhendo abacates...\n", id);
```

```
    sleep(id%5+5);

    pthread_mutex_lock(&mutex);


    // Caso nao hajam mais caixas para troca
    // ou navio nao esta em A
    // ou anciao quer trocar caixa
    // o habitante espera

    while( estoque_B == 0 ||
           !navio_em_a    ||
           ancioes_querem_A){

    printf("Habitante %d (ilha A): esperando...\n", id);
      pthread_cond_wait(&habitantes_A, &mutex);
    }

    // Habitante abastece navio navio
    estoque_A++;
    estoque_B--;
    printf("Habitante %d (ilha A):\
    abastece navio com fruta (A:%d; B:%d)\n",
           id, estoque_A, estoque_B);


    // Caso nao hajam mais caixas para troca
    // ele    liberado para zarpar
    if ( estoque_B == 0){
      pthread_cond_signal(&zarpar_A);
    }

    pthread_mutex_unlock(&mutex);
    sleep(3);
  }
}
```

Já as threads que representam os anciões atuam de maneira similar as threads que representam os anciões. Uma das diferenças está no fato de que, após realizar o processo de colheita, os anciões sinalizam que terminaram sua colheita e estão indo em direção ao navio por meio do incremento da variável inteira `ancioes_querem_A`. Consequentemente, a variável deve ser decrementada quando o ancião termina de realizar sua troca. Além disso, ao checar as condições para que possa realizar a troca, o ancião não deve se certificar de que existe mais algum ancião que deseja realizar a troca, visto que ele tem prioridade.

Função executada pelas threads que representam os anciões de A.

```
void * f_anciaoA(void *arg){

  int id = *((int*)arg);
  while(1){

    // Anciao colhe abacates
    printf("Anciao    %d (ilha A):\
    colhendo abacates...\n", id);
    sleep(id%5+5);

    pthread_mutex_lock(&mutex);
    printf("Anciao    %d (ilha A):\
    eu ja colhi meu abacate\n", id);
    ancioes_querem_A++;

    // Caso nao hajam mais caixas para troca
    // ou navio nao esta em A
    // o anciao espera
    while( !navio_em_a || estoque_B == 0){
    printf("Anciao    %d (ilha A): esperando...\n", id);
      pthread_cond_wait(&habitantes_A, &mutex);
    }

    // Anciao abastece navio navio
    estoque_A++;
    estoque_B--;
    printf("Anciao    %d (ilha A):\
    abastece navio com fruta (A:%d; B:%d)\n",
           id, estoque_A, estoque_B);

    ancioes_querem_A--;


    // Caso nao hajam mais caixas para troca
    // ele    liberado para zarpar
    if ( estoque_B == 0){
      pthread_cond_signal(&zarpar_A);
    }

    pthread_mutex_unlock(&mutex);
    sleep(3);
  }
}
```

Os passos a serem realizados pelas threads que representam os habitantes e

anciões da ilha são análogos aos discutidos acima, porém deve ser feito o uso das variáveis de condição `habitantes_B` e `zarpar_B`, além da variável inteira `ancioes_querem_B`.

4 Conclusão

Com base na implementação do algoritmo apresentado, foi constatado que o problema pôde ser solucionado sem a ocorrência de condições de corrida. Sendo assim, o acesso à memória não é feito por mais de uma thread ao mesmo tempo, ao passo de que ainda há concorrência entre os processos quando os mesmos não estão acessando regiões criticas. Dessa forma, é possível que os habitantes das ilhas realizem suas colheitas de forma simultânea.

5 Referências

Referências

[1] Reading 23: Locks and Synchronization. URL: `https://web.mit.edu/6.005/www/fa15/classes/23-locks/`.

[2] IBM Documentation, January 2023. URL: `https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/aix/7.2?topic=programming-using-condition-variables`.

[3] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[4] David W. Bustard. Concepts of concurrent programming. 1990.

[5] Brian Harvey and Matthew Wright. *Simply Scheme (2nd Ed.): Introducing Computer Science*. MIT Press, Cambridge, MA, USA, 1999.

[6] Stephen Samuel and Stefan Bocutiu. *Programming Kotlin*. Packt Publishing, 2017.