CIC202 - Programação Concorrente: Trabalho 1

Sincronização de Processos: A Caverna dos Sableyes

Guilherme da Rocha Cunha - 221030007

2024.1

UnB

Sumário

1 Introdução

O trabalho consiste na aplicação dos conhecimentos adquiridos na disciplina "CIC202 - Programação Concorrente". Nele é apresentado o problema de comunicação de processos atráves de uma memória compartilhada, "A Caverna dos Sableyes", e sua solução utilizando a biblioteca POSIX Pthereads na linguagem C.

2 Formalização do Problema: A Caverna dos Sableyes

Sableyes são uma espécie de pokemon de pele roxa, garras e dentes afiados, que se alimentam de pedras preciosas e moram no interior de cavernas.

Figure 1: Sableye

É comum deles perambularem pelas câmaras das cavernas em que habitam em busca de joias e carbinks (suas presas), principalmente depois de um terremoto. Após um terremoto, as câmaras das cavernas têm uma chance de revelar joias que até então estavam enterradas ou presas no teto da caverna.

O problema consiste na simulação da busca dos sableyes por alimento. Há n sableyes e m câmaras na caverna, onde cada câmara possui um quantidade j_i de joias preciosas reveladas e uma capacidade q_i ($1 \leq i \leq m$).

Se um sableye está na i-ésima câmara, ele vai tentar comer até j_i joias. No caso de não houver joias ou espaço na câmara, ele vai embora e tenta procurar por comida em outro lugar.

Em um determinado momento, ocorrerá um terremoto na caverna. Para cada câmara i, o terremoto irá revelar k_i novas joias. Enquanto estiver ocorrendo um terremoto, nenhum sableye estará procurando por comida.

3 Descrição do Algoritmo

Para resolver este problema, utilizaremos n threads para representar cada sableye e 1 representando a caverna, para realizar a ação do terremoto.

3.1 Thread Caverna

Esta única thread será responsável por executar a função `terremoto()` na simulação.

No início, a função executa um `sleep()` de duração entre 10 a 19 segundos, com o intuito de permitir que os sableye procurem por joias nas câmaras. Uma vez que este `sleep()` termina, a função levanta a flag `comecoTerremoto`, indicando que ocorrerá um terremoto, fazendo com que os sableyes não procurem por comida durante a execução da função.

Com a flag ativada, a função tranca o mutex `mutex` e começa a atualizar as quantidades de joias em cada câmara.

Após isso, a flag é desativada e é executado um `pthread_cond_brodcast()` na variável de condição `sableye_cond`, liberando a busca dos sableyes nas câmaras.

```
// terremoto
comecoTerremoto = true;
// regiao critica
pthread_mutex_lock(&mutex);
        // terremoto revelando joias
        for(int i = 0; i < NUMCAM; i++) {
        int joiasReveladas = ((int) rand() % 10) +1;
        qntJoias[i] += joiasReveladas;
        }

        // fim do terremoto
        comecoTerremoto = false;

        // liberando os sableyes
        pthread_cond_broadcast(&sableye_cond);
pthread_mutex_unlock(&mutex);
```

Por fim, o mutex é liberado e o algoritmo se repete.

3.2 Thread Sableye

Primeiramente, se estiver ocorrendo um terremoto, todos os sableyes irão esperá-lo terminar. Para isso, utilizamos a variável de condição descrita anteriormente `sableye_cond` junto com a função `pthread_cond_wait()`.

```
while(comecoTerremoto) {
        pthread_cond_wait(&sableye_cond, &mutex);
}
```

Uma vez com o terremoto encerrado, as threads executaram um `sleep()` para simular a ação de procura de alimento, além de "bagunçar" a sincronia das threads.

De forma aleatória, as threads sorteiam a câmara a ser explorada. As câmaras serão semáforos, onde `camaras[NUMCAM]` será responsável por contralar

o acesso da memória (região crítica do código) compartilhada da i-ésima camara e `capCamaras[NUMCAM]` será responsável por controlar a capacidade.

Uma thread executará a função `sem_trywait()` em `capCamaras[idx]`, ou seja, um sableye tentará entrar na câmara `idx`. Se o retorno for 0, ele entrará na câmara, e caso contrário ele irá procurar em outro lugar.

Uma vez na câmara `idx`, é feito a operação DOWN no semáforo `camara[idx]` de capacidade 1 para verificar a quantidade de joias presentes na câmara atual. Caso houver joias, é sorteado um número j_s ($1 \leq j_s \leq j_{idx}$), isto é, o número de joias comidas pelo sableye, e assim, retirado da quantidade total de joias presentes no local.

Finalmente, os semáforos `camaras[idx]` e `capCamaras[idx]` sofrem a operação UP, onde `sem_post(&capCamaras)` representa o sableye saindo da câmara onde se encontra, dando espaço para o próximo.

```
// sorteia uma camara
int idx = ((int) rand()) % NUMCAM;

// regiao critica
// verifica se ha espaco na camara
if(sem_trywait(&capCamaras[idx]) == 0) {
        sem_wait(&camaras[idx]);
        if(qntJoias[idx]) {
                int joiasComidas = ((int) rand() % qntJoias[idx]) +1;

                // comendo
                sleep(((int) rand() % 3)+1);

                qntJoias[idx] -= joiasComidas;

        } else {
                // nao ha joias para comer
        }
        sem_post(&camaras[idx]);
        sem_post(&capCamaras[idx]);
} else {
        // camara cheia
}
```

4 Conclusão

Fica evidente o quão poderoso e importante a programação concorrente realmente é, visto que ao utilizar esta técnica problemas como este têm suas complexidades e custo operacional bastante reduzidos, se compararmos com programas de um único fluxo de execução que tentam resolver o mesmo problema.

5 Referências

- Ben-Ari, M., Principles of Concurrent and Distributed Programming, Prentice Hall, 2a ed., 2006.
- Gregory Andrews, Concurrent Programming: Principles and Practice, Addison-Wesley, 1991, ISBN 0805300864.
- Breshears, C., The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications, O'Reilly, 2009.
- stackoverflow.com
- cs.cs.menu.edu
- bulbapedia.bulbagarde.net

Repositório do trabalho: link.