TRABALHO FINAL DE PROGRAMAÇÃO CONCORRENTE

RELATÓRIO

Nome: Gustavo Mello Tonnera

Disciplina: Programação Concorrente

Professor: Eduardo Adilio Pelinson Alchieri

Semestre: 2024.1

1. INTRODUÇÃO

A Programação Concorrente é um paradigma de programação que busca resolver problemas que necessitam da execução simultânea de diferentes partes de uma mesmo programa. A disciplina de Programação Concorrente ofertada pelo professor Eduardo Alchieri almeja introduzir os estudantes de computação da UnB (Universidade de Brasília) aos conceitos desse paradigma extremamente importante nos dias de hoje.

Durante as aulas, o professor explica os principais conceitos da Programação Concorrente, como locks, variáveis de condição, sinais, condições de corrida etc. Para avaliar o conhecimento adquirido pelos estudantes na disciplina, o professor propôs duas provas e um trabalho.

O trabalho consiste no desenvolvimento de um problema que envolva programação concorrente e de sua solução. O problema precisa conter comunicação entre processos através de memória compartilhada com condições de corrida, situações em que dois ou mais processos acessam dados compartilhados e o resultado final depende de quem executa e quando executa. A solução para o problema precisa conter mecanismos de sincronização de processos estudados na sala de aula, como locks, variáveis de condição e semáforos, e precisa utilizar a biblioteca POSIX Pthreads.

O problema proposto pelo aluno Gustavo M. Tonnera foi chamado de "O problema das picaretas". A formalização e a solução do problema são explicadas nas próximas seções do relatório.

2. O PROBLEMA DAS PICARETAS

A cidade Vale das Minas é uma cidade que se sustenta devido à exportação de minérios extraídos de duas minas perto da cidade: uma mina de diamantes e uma mina de ouro. Os moradores acordam cedo para trabalhar na mina e vender os minérios para os comerciantes que passam pela cidades. Além disso, como os minérios encontrados nas minas possuem grande dureza, as picaretas utilizadas pelos mineradores possuem uma chance de 33% de quebrarem durante a extração dos minérios.

Recentemente, um grupo de ladrões invadiu a cidade e roubou grande parte das picaretas dos mineradores. Como a cidade é pequena, ela não possui

ferreiros e os mineiros precisarão compartilhar as picaretas para que a economia da cidade não tão afetada.

Desenvolva um algoritmo que simule um número M1 de mineiros trabalhando na mina de ouro, um número M2 de mineiros trabalhando na mina de diamantes e um número P de picaretas, com P < M1+M2. Como diamantes são mais valiosos do que ouro, os mineiros da mina de diamantes possuem prioridade para utilizarem as picaretas remanescentes do roubo. Caso um mineiro tente pegar uma picareta, mas não tiver nenhuma picareta disponível, ele dormirá até que outro mineiro termine de usar a picareta e o acorde.

3. SOLUÇÃO DO PROBLEMA

Para resolver o problema, foi utilizado locks e variáveis de condição como mecanismos de sincronização de processos. O arquivo com o código da solução está armazenado em um repositório do GitHub, o qual pode ser acessado pelo seguinte link: https://github.com/GMTonnera/unb-prog_concorrente. Para explicar a solução desenvolvida, dividiu-se o código em 4 partes: Declaração de variáveis globais, Função main, Thread do tipo mineiro de ouro e Thread do tipo mineiro de diamante.

3.1 DECLARAÇÃO DE VARIÁVEIS GLOBAIS

Nessa etapa do código, o número de mineiros e as variáveis compartilhadas entre os processos são declarados. Além disso, os locks e as varáveis de condição também são inicializados.

```
6   #define MINEIROS_OURO 15
7   #define MINEIROS_DIAMANTE 15
8
9   int picaretas = 10;
10  int m_ouro_quer = 0;
11  int m_diamante_quer = 0;
12
13  pthread_mutex_t mutex_picaretas = PTHREAD_MUTEX_INITIALIZER;
14  pthread_cond_t cond_mineiro_ouro = PTHREAD_COND_INITIALIZER;
15  pthread_cond_t cond_mineiro_diamante = PTHREAD_COND_INITIALIZER;
16
17  void * mineiroOuro(void* id);
18  void * mineiroDiamante(void* id);
```

Na imagem acima, observa-se a definição do número de mineiros de ouro (*MINEIROS_OURO*) e do número de mineiros de diamante (*MINEIROS_DIAMANTE*) com o valor de 10, nas linhas 6 e 7, respectivamente.

Observa-se, também, a declaração do número de picaretas que podem ser usadas (*picaretas*) na linha 9, do número de mineiros de ouro que estão esperando uma picareta para trabalhar (*m_ouro_quer*) na linha 10 e do número de mineiros de diamante que estão esperando uma picareta para trabalhar (*m_diamante_quer*) na linha 11. Logo abaixo, vê-se a inicialização de um lock (linha 13), de duas variáveis de condição (linhas 14 e 15) e as declarações das funções que as threads dos tipos de mineiro executam (linhas 17 e 18).

O lock *mutex_picaretas* é responsável por impedir que dois ou mais processos acessem as varáveis *picaretas* simultaneamente. As variáveis de condição *cond_mineiro_ouro* e *cond_mineiro_diamante* são responsáveis por fazer os mineiros de ouro e os mineiros de diamante dormirem e acordarem, respectivamente.

3.2 FUNÇÃO MAIN

A função main é a função responsável por criar as threads do tipo mineiro e do tipo ferreiro.

```
20  void main(int argc, char* argv[]) {
21      int erro;
22      int *id;
23
24      pthread_t tmo[MINEIROS_OURO];
25
26      // Criar threads dos mineiradores de ouro
27      for (int i = 0; i < MINEIROS_OURO; i++) {
28          id = (int *) malloc(sizeof(int));
29          *id = i;
30          erro = pthread_create(&tmo[i], NULL, mineiroOuro, (void *) (id));
31
32          if(erro) {
33              printf("erro na criacao do thread %d\n", i);
34              exit(1);
35          }
36      }
37
38      pthread_t tmd[MINEIROS_DIAMANTE];
39
40      // Criar threads dos mineiradores de diamante
41      for (int i = 0; i < MINEIROS_DIAMANTE; i++) {
42          id = (int *) malloc(sizeof(int));
43          *id = i;
44          erro = pthread_create(&tmd[i], NULL, mineiroDiamante, (void *) (id));
45
46          if(erro) {
47              printf("erro na criacao do thread %d\n", i);
48              exit(1);
49          }
50      }
51
52      pthread_join(tmo[0], NULL);
53  }
```

Na linha 21, a variável *erro* é declarada, cuja função é detectar erros na criação das threads. Na linha 22, a varável *id* é declarada, cuja função é

armazenar um identificador único das threads. Na linha 23, um array de threads (*tmo*) é declarado com o tamanho *MINEIROS_OURO* para armazenar as threads do tipo mineiro de ouro.

Nas linhas 27 a 36, um loop cria todas as threads do tipo mineiro de ouro e verifica se ocorreu algum erro na criação de alguma das threads. Se ocorreu, a mensagem “erro na criação do thread x”, onde x é o id da thread, é exibida no terminal e o programa é encerrado. Se não ocorreu erro, as threads são armazenados no array de threads *tmo*.

Na linha 38, outro array de threads (*tmd*) é declarado, cuja função é armazenar as threads do tipo mineiro de diamante. Nas linhas 41 a 50, um loop cria todas as threads do tipo mineiro de diamante e verifica se ocorreu algum erro na criação de alguma das threads. Se ocorreu, a mensagem “erro na criação do thread x”, onde x é o id da thread, é exibida no terminal e o programa é encerrado. Se não ocorreu erro, as threads são armazenados no array de threads *tmd*. Por fim, na linha 52, a thread que está executando a função main realiza uma operação de join com a thread de index 0 do array *tmo*.

3.3 THREAD DO TIPO MINEIRO DE OURO

Threads do tipo mineiro de ouro são responsáveis por simular o comportamento dos mineiros que extraem ouro da mina de ouro da cidade do problema. Essas threads executam a função *mineiroOuro* presente na imagem abaixo.

```
56  void * mineiroOuro(void* id) {
57      int m_id =  *(int *)(id);
58
59      while (1) {
60          pthread_mutex_lock(&mutex_picaretas);
61              m_ouro_quer++;
62              while (picaretas == 0 || m_diamante_quer >= MINEIROS_DIAMANTE / 2) {
63                  pthread_cond_wait(&cond_mineiro_ouro, &mutex_picaretas);
64              }
65              m_ouro_quer--;
66              picaretas--;
67              printf("Mineiro de Ouro %d: pegou uma picareta. Picaretas = %d.\n", m_id, picaretas);
68          pthread_mutex_unlock(&mutex_picaretas);
69
70          printf("Mineiro de Ouro %d: trabalhando.\n", m_id);
71          sleep(5);
72
73          pthread_mutex_lock(&mutex_picaretas);
74              picaretas++;
75              printf("Mineiro de Ouro %d: devolveu a picareta. Picaretas = %d.\n", m_id, picaretas);
76              pthread_cond_broadcast(&cond_mineiro_ouro);
77              pthread_cond_broadcast(&cond_mineiro_diamante);
78          pthread_mutex_unlock(&mutex_picaretas);
79          printf("Mineiro de Ouro %d: dormindo.\n", m_id);
80          sleep(10);
81      }
82  }
```

Na linha 57, declara-se a variável *m_id*, a qual armazena o id da thread. Na linha 59, declara-se um while loop que será executado até que o programa seja interrompido. Na linha 60, a thread ativa o lock de *mutex_picaretas* para ler e modificar a variável *picaretas*. Na linha 61, incremente-se em um o valor da variável *m_ouro_quer*, indicando que mais um mineiro de ouro está esperando por uma picareta. Na linha 62, verifica-se se o número de picaretas disponíveis para uso é igual a 0 ou se o número de mineiros de diamantes que estão aguardando uma picareta é maior ou igual ao número total de mineiros de diamantes dividido por dois. Se não, a thread executa um wait usando a variável de condição *cond_mineiros_ouro* (linha 63), se sim decrementa-se em um o valor da variável *m_ouro_quer* (linha 65) e da variável *picaretas* (linha 66). Em seguida, na linha 68, a thread desativa o lock *mutex_picaretas*.

A verificação do número de mineiro de diamante esperando uma picareta é feita para implementar a prioridade que os mineiros de diamante possuem sobre os mineiros de ouro descrita no problema. Essa maneira de implementar a prioridade foi escolhida para evitar starvation de threads do tipo mineiro de ouro no programa. Caso verificássemos se o número de mineiros de diamante esperando uma picareta é maior que zero, poderia acontecer de nenhum mineiro de ouro conseguir minerar.

O sleep da linha 71 é usado para simular o mineiro trabalhando. Na linha 73, ativa-se novamente o lock *mutex_picaretas*. Na linha 74, incrementa-se o número da variável *picaretas* em um, indicando que o mineiro de ouro já terminou de trabalhar. Nas linhas 76 e 77, emite-se sinais de broadcast para as threads do tipo mineiro de ouro e do tipo mineiro de diamante, a fim de acordar todos os mineiros, já que uma picareta está disponível para uso. Na linha 78, o lock *mutex_picaretas* é desativado. O sleep da linha 80 simula o descanso do mineiro de ouro.

3.4 THREAD DO TIPO MIENEIRO DE DIAMANTE

Threads do tipo mineiro de diamante são responsáveis por simular o comportamento dos mineiros que extraem diamante da mina de diamantes da cidade do problema. Essas threads executam a função *mineiroDiamante* presente na imagem abaixo.

```
84  void * mineiroDiamante(void* id) {
85      int m_id =  *(int *)(id);
86
87      while (1){
88          pthread_mutex_lock(&mutex_picaretas);
89              m_diamante_quer++;
90              while (picaretas == 0) {
91                  pthread_cond_wait(&cond_mineiro_diamante, &mutex_picaretas);
92              }
93              m_diamante_quer--;
94              picaretas--;
95              printf("Mineiro de Diamante %d: pegou uma picareta. Picaretas = %d.\n", m_id, picaretas);
96          pthread_mutex_unlock(&mutex_picaretas);
97
98          printf("Mineiro de Diamante %d: trabalhando.\n", m_id);
99          sleep(5);
100
101         pthread_mutex_lock(&mutex_picaretas);
102             picaretas++;
103             printf("Mineiro de Diamante %d: devolveu a picareta. Picaretas = %d.\n", m_id, picaretas);
104             pthread_cond_broadcast(&cond_mineiro_ouro);
105             pthread_cond_broadcast(&cond_mineiro_diamante);
106         pthread_mutex_unlock(&mutex_picaretas);
107         printf("Mineiro de Diamante %d: dormindo.\n", m_id);
108         sleep(10);
109     }
110 }
```

Na linha 85, declara-se a variável *m_id*, a qual armazena o id da thread. Na linha 87, declara-se um while loop que será executado até que o programa seja interrompido. Na linha 88, a thread ativa o lock de *mutex_picaretas* para ler e modificar a variável *picaretas*. Na linha 89, incremente-se em um o valor da variável *m_diamante_quer*, indicando que mais um mineiro de diamante está esperando por uma picareta. Na linha 90, verifica-se se o número de picaretas disponíveis para uso é igual a 0. Se não, a thread executa um wait usando a variável de condição *cond_mineiros_diamante* (linha 91), se sim decrementa-se em um o valor da variável *m_diamante_quer* (linha 93) e da variável *picaretas* (linha 94). Em seguida, na linha 96, a thread desativa o lock *mutex_picaretas*.

O sleep da linha 99 é usado para simular o mineiro trabalhando. Na linha 101, ativa-se novamente o lock *mutex_picaretas*. Na linha 102, incrementa-se o número da variável *picaretas* em um, indicando que o mineiro de diamante já terminou de trabalhar. Nas linhas 104 e 105, emite-se sinais de broadcast para as threads do tipo mineiro de ouro e do tipo mineiro de diamante, a fim de acordar todos os mineiros, já que uma picareta está disponível para uso. Na linha 106, o lock *mutex_picaretas* é desativado. O sleep da linha 108 simula o descanso do mineiro de diamante.

4. CONCLUSÃO

A programação concorrente desempenha um papel vital na resolução de problemas complexos que envolvem a execução simultânea de múltiplas tarefas

e o uso coordenado de recursos limitados. Em muitos sistemas modernos, como no exemplo de Vale das Minas, onde a quantidade de picaretas disponíveis era insuficiente para atender a todos os mineiros, a programação concorrente se mostrou indispensável.

Por meio de técnicas como locks e variáveis de condição, foi possível garantir que os recursos fossem compartilhados de forma eficiente e ordenada, evitando conflitos e maximizando a produtividade. Os locks asseguraram que apenas um mineiro acessasse uma picareta por vez, prevenindo o uso simultâneo e, consequentemente, erros no processo. Já as variáveis de condição foram essenciais para controlar a espera dos mineiros que não encontravam picaretas disponíveis, permitindo que eles fossem notificados assim que o recurso fosse liberado.

Esse tipo de coordenação e gerenciamento só é viável graças à programação concorrente, que possibilita a criação de sistemas capazes de lidar com as complexidades de tarefas paralelas. Em resumo, a programação concorrente é crucial para a construção de soluções eficientes e robustas em situações em que o desempenho e a correta alocação de recursos são determinantes para o sucesso do sistema.

5. REFERÊNCIAS

Slides apresentados em sala de aula.

Videoaulas disponibilizadas pelo professor.

Códigos disponibilizados pelo professor.

https://docs.oracle.com/cd/E26502_01/html/E35303/tlib-1.html

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html