**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 e Thread do tipo ferreiro.

**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.

**Texto

Descrição gerada automaticamente**

Na imagem acima, observa-se a definição do número de mineiros (*MINEIROS*) com o valor de 10 e a definição da chance da chance de a picareta quebrar (*CHANCE*) com o valor de 33, nas linhas 6 e 7. Observa-se, também, a declaração do número de picaretas que podem ser usadas (*picaretas*) e do número total de picaretas (*total\_picaretas*) com o valor de 3, nas linhas 9 e 10. Logo abaixo, vê-se a inicialização dos locks (linhas 12 e 13), das variáveis de condição (linhas 14 e 15) e as declarações das threads do tipo mineiro (linha 17) e do tipo ferreiro (linha 18).

Os locks *mutex\_picaretas* e *mutex\_total\_picaretas* são responsáveis por impedir que dois ou mais processos acessem as varáveis *picaretas* e *total\_picaretas* simultaneamente, respectivamente. As variáveis de condição *cond\_mineiros* e *cond\_ferreiro* são responsáveis por fazer os mineiros e os ferreiros 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.

**Texto

Descrição gerada automaticamente**

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 é declarado com o tamanho *MINEIROS* para armazenar as threads do tipo mineiro.

Nas linhas 26 a 35, um loop cria todas as threads do tipo mineiro 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 *tm*.

Na linha 37, a thread do ferreiro é declarada. Na linha 38, a thread do ferreiro é criada. Nas linhas 40 a 42, verifica-se se ocorreu algum erro na criação da thread do ferreiro. Se ocorreu algum erro, a mensagem “erro na criação do thread x”, onde x é o id da thread, é exibida no terminal e o programa é encerrado. Na linha 45, a thread que está executando a função realiza uma operação de join com a thread do tipo ferreiro, uma vez que só pode existir uma única thread do tipo ferreiro.

**3.3 THREAD DO TIPO MINEIRO**

Threads do tipo mineiro são responsáveis por simular o comportamento dos mineiros do problema. Essas threads executam a função *mineiro* presente na imagem abaixo.

**Texto

Descrição gerada automaticamente**

Na linha 50, declara-se a variável *m\_id*, a qual armazena o id da thread. Na linha 51, declara-se a variável *picareta\_quebrou*, cuja função é armazenar o valor 0 se a picareta não quebrou durante o trabalho do minerador e armazenar 1 se a picareta quebrou durante o trabalho do mineiro. Na linha 52, determinamos um seed para a thread, a fim de evitar que outras threads gerem os mesmo números aleatórios.

Na linha 53, declara-se um while loop que será executado até que o programa seja interrompido. Na linha 54, atribuímos à variável *picareta\_quebrou* o valor 0. Na linha 55, a thread ativa o lock de *mutex\_picaretas* para ler a variável *picaretas*. Na linha 56, verifica-se se existe picaretas disponíveis para uso. Se não a thread executa um wait usando a variável de condição cond\_mineiros (linha 57), se sim diminui-se em um o valor da variável picaretas (linha 59) e a thread desativa o lock *mutex\_picaretas*.

O sleep da linha 64 é usado para simular o mineiro trabalhando. Nas linhas 65 a 67, gera-se um número aleatório entre 0 e 99 inclusos e verifica-se se esse número é menor ou igual a chance de a picareta quebrar. Se sim, atribui-se o valor 1 à variável *picareta\_quebrou*.

Na linha 69, ativa-se novamente o lock *mutex\_picaretas*. Nas linhas 70 a 79, verifica-se se a picareta quebrou. Se não, o mineiro devolve a picareta somando em um o valor da variável *picaretas* (linha 71) e damos um broadcast nas outras threads do tipo mineiro (linha 73), acordando os mineiros que estiverem dormindo, uma vez que uma picareta está disponível para uso. Se sim, ativamos o lock *mutex\_total\_picaretas* (linha 75), diminuímos o valor da variável total\_picaretas em um (linha 76), desativamos o lock mutex\_total\_picaretas (linha 78) e realizamos um signal para a thread do tipo ferreiro (linha 79), para que, caso ela esteja dormindo, ele volte a ser executada para produzir novas picaretas.

**3.4 THREAD DO TIPO FERREIRO**

Como especificado no problema, existe apenas uma única thread responsável por simular o comportamento do ferreiro. Essa thread é a thread do tipo ferreiro, a qual executa a função presente na imagem abaixo.

Texto

Descrição gerada automaticamente

Na linha 88, declara-se um while loop que será executado até que o programa seja interrompido. Na linha 89, ativa-se o lock *mutex\_total\_picaretas* para acessar a variável total

**4. CONCLUSÃO**

**5. REFERÊNCIAS**