Logotipo, nome da empresa

Descrição gerada automaticamente

Mateus Boletta

Lucas Albino

**Trabalho de Pipes e Threads**

Poços de Caldas

2024

Mateus Boletta

Lucas Albino

**Trabalho de Pipes e Threads**

Tem como objetivo desenvolver um código em C (puro), utilizando o GCC Linux onde dois (ou mais) sub processos intercalam suas execuções e trocam mensagens através da IPC Pipes.

Como objetivo secundário, essa implementação deve considerar alguma aplicação e deve haver também a utilização de, pelo menos, uma thread.

Professor : João Carlos de Moraes Morselli Junior

Poços de Caldas

2024

**SUMÁRIO**

|  |  |
| --- | --- |
| **1 INTRODUÇÃO** | 3 |
| * 1. **Ferramentas Utilizadas**      1. *Pipes*      2. Processos      3. *Threads*      4. *Mutex* | 3  3  3  3  4 |
| 1. **FUNCIONAMENTO LÓGICO**    1. **Criação do *Pipe***    2. ***Fork* do Subprocesso** | 4  4  4 |
| * + 1. Subprocesso     2. Processo Principal   1. **Função do Subprocesso**   2.3.1 O que ocorre   * 1. **Execução das *Threads***   2.4.1 Verifica o comando recebido: | 5  5  6  6  7  7 |
| **3 EXECUÇÃO** | 8 |
| * 1. **Preparando o Ambiente** | 8 |
| 3.1.1 Verificar e Instalar o GCC  **3.2 Compilando e Executando**  3.2.1 Compilando  3.2.2 Executando  **3.3 Funcionamento** | 8  9  9  9  10 |
|  |  |
|  |  |

#### **1 INTRODUÇÃO**

Neste trabalho optamos por confeccionar um Projeto de Simulação Bancária com Concorrência ou como optamos em chamar de PSBC bank.

No PSBC bank, implementamos um sistema bancário simples utilizando conceitos fundamentais da programação concorrente em C. O objetivo é demonstrar como utilizar ***pipes***, **processos** e ***threads*** para criar uma aplicação que simule operações bancárias básicas de forma concorrente e segura.

* 1. **Ferramentas Utilizadas**

1.1.1 *Pipes:*

São utilizados para comunicação entre o processo principal e o subprocesso. Os *pipes* permitem a troca de dados entre processos, sendo essenciais para enviar comandos do processo principal para o subprocesso que executa as operações bancárias

* + 1. Processos:

Utilizamos o *fork()* para criar um subprocesso. O processo principal é responsável por ler os comandos do usuário e enviá-los através do *pipe*, enquanto o subprocesso lê esses comandos e cria *threads* para processá-los.

* + 1. *Threads*:

Para processar cada comando de forma concorrente, utilizamos *threads*. Cada comando recebido pelo subprocesso é tratado por uma nova *thread*, permitindo que múltiplas operações bancárias sejam realizadas simultaneamente. As *threads* são gerenciadas pela função *pthread\_create().*

* + 1. *Mutex*:

Um *mutex* (*pthread\_mutex\_t*) é utilizado para garantir acesso seguro ao saldo da conta bancária compartilhada. O *mutex* impede condições de corrida ao garantir que apenas uma *thread* possa modificar o saldo da conta por vez.

**2 FUNCIONAMENTO LÓGICO**

**2.1 Criação do Pipe**:

O programa começa criando um *pipe*, que será utilizado para a comunicação entre o processo principal e o subprocesso. O *pipe* cria dois *file descriptors*: um para a leitura (*pipe\_fd[0])* e outro para a escrita (*pipe\_fd[1*])..

Texto

Descrição gerada automaticamente

Se a criação do *pipe* falhar, o programa imprime uma mensagem de erro e encerra.

* 1. ***Fork* do Subprocesso**:

Em seguida, o programa utiliza *fork*() para criar um subprocesso. O *fork*() retorna o PID do subprocesso ao processo pai e zero ao subprocesso. Se *fork(*) falhar, imprime uma mensagem de erro e encerra.

Uma imagem contendo Forma

Descrição gerada automaticamente

* + 1. Subprocesso:

O subprocesso fecha a extremidade de escrita do *pipe* (*pipe\_fd[1]*), pois só precisa ler comandos do processo principal. Depois, chama *banking\_subprocess()* para processar esses comandos.

Tela preta com letras brancas

Descrição gerada automaticamente

2.2.2 Processo Principal:

O processo principal fecha a extremidade de leitura do *pipe* (*pipe\_fd[0]*), pois só precisa enviar comandos ao subprocesso. Em seguida, entra em um loop que lê comandos do usuário e os escreve no *pipe*.

Texto

Descrição gerada automaticamente

* 1. **Função do Subprocesso**:

A função *banking\_subprocess*() é chamada pelo subprocesso para ler comandos do *pipe* e criar threads para processá-los.

Texto

Descrição gerada automaticamente

2.3.1 O que ocorre:

* Lê comandos do *pipe* com *read(*). Se a leitura falhar ou o *pipe* for fechado *(bytes\_read <= 0)*, o *loop* é encerrado.
* Se o comando for "sair", imprime uma mensagem e sai do loop.
* Para outros comandos, cria uma thread usando *pthread\_create(),* passando a função *handle\_request()* como argumento. Usa *strdup()* para duplicar a *string* de comando, garantindo que cada thread tenha sua própria cópia.
* Usa *pthread\_detach()* para assegurar que os recursos da thread sejam liberados automaticamente após a execução
  1. **Execução das *Threads***:

Cada thread criada executa a função *handle\_request()*, que processa o comando recebido.

Texto

Descrição gerada automaticamente

Faz o *cast* do argumento *arg* para *char*\*, que contém o comando e usa um *mutex* (*pthread\_mutex\_lock*()) para proteger o acesso ao saldo da conta compartilhada.

2.4.1 Verifica o comando recebido:

* **Depositar**: Adiciona o valor especificado ao saldo.
* **Sacar**: Verifica se há fundos suficientes e subtrai o valor do saldo, se possível.
* **Saldo**: Imprime o saldo atual.
* **Comando Desconhecido**: Imprime uma mensagem de erro.

Desbloqueia o *mutex* (*pthread\_mutex\_unlock())* e libera a memória alocada para o comando (*free(request*)).

1. **EXECUÇÃO**

**3.1 Preparando o Ambiente**

Para executar a aplicação em um GCC no Linux conforme solicitado, optamos por utilizar uma máquina virtual do Linux em nossas máquinas. Utilizamos do *Oracle VM Virtual Box* para emular uma máquina do *Linux Ubuntu*

.

Para conseguirmos executar no nosso terminal do Linux será necessário preparar a máquina.

3.1.1 Verificar e Instalar o GCC:

Para Verificar digite **gcc –version**:

Texto

Descrição gerada automaticamente

Como pode ver, não tínhamos o gcc, mas para isso precisamos instalar o gcc digitando:

**sudo apt update**

**sudo apt install gcc**

Texto

Descrição gerada automaticamente

* 1. **Compilando e Executando**

Após Ter o GCC instalado no Linux, precisamos compilar o código no terminal com o seguinte comando: (PSBCbank é o nome do arquivo)

* + 1. Compilando:

**gcc -o PSBCbank PSBCbank.c -pthread**

**Texto

Descrição gerada automaticamente**

* + 1. Executando:

**Para Executar digite:**

**./PSBCbank**

**Texto

Descrição gerada automaticamente**

* 1. **Funcionamento**

Texto

Descrição gerada automaticamente

Vamos analisar um exemplo detalhado da execução do nosso programa bancário. O programa possui quatro funções principais: depositar, sacar, verificar saldo e sair. A execução do programa demonstra a interação do usuário com essas funções. Inicialmente, o saldo é definido como R$ 1.000,00. Primeiramente, realizamos um saque de R$ 500,00. O saldo é atualizado e o novo saldo é exibido como R$ 500,00. Em seguida, depositamos R$ 1.000,00 na conta. O saldo é novamente atualizado e o novo saldo é exibido como R$ 1.500,00. Para verificar o saldo, usamos o comando "saldo". O saldo atual é exibido como R$ 1.500,00. Para garantir que o sistema não permite saques maiores que o saldo disponível, tentamos sacar R$ 2.000,00. O sistema corretamente detecta a insuficiência de fundos e exibe uma mensagem apropriada, sem alterar o saldo. Finalmente, usamos o comando "sair" para encerrar o programa. O saldo é manipulado corretamente e os erros, como a tentativa de saque com fundos insuficientes, são tratados adequadamente.

Além disso, o programa possui uma região crítica, protegida por um *mutex*, que garante a integridade das operações de saque e depósito. Essa região crítica impede que saques simultâneos na mesma conta causem inconsistências no saldo. O *mutex* garante que apenas uma *thread* por vez possa acessar e modificar o saldo da conta, evitando condições de corrida e garantindo que as operações financeiras sejam realizadas de maneira segura e consistente. Desta forma, o programa assegura que os usuários não enfrentem problemas decorrentes de operações simultâneas conflitantes.