



PUC Minas
Poços de Caldas

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
1.1	Ferramentas Utilizadas	3
1.1.1	<i>Pipes</i>	3
1.1.2	Processos	3
1.1.3	<i>Threads</i>	3
1.1.4	<i>Mutex</i>	4
2	FUNCIONAMENTO LÓGICO	4
2.1	Criação do <i>Pipe</i>	4
2.2	<i>Fork</i> do Subprocesso	4
2.2.1	Subprocesso	5
2.2.2	Processo Principal	5
2.3	Função do Subprocesso	6
2.3.1	O que ocorre	6
2.4	Execução das <i>Threads</i>	7
2.4.1	Verifica o comando recebido:	7
3	EXECUÇÃO	8
3.1	Preparando o Ambiente	8
3.1.1	Verificar e Instalar o GCC	8
3.2	Compilando e Executando	9
3.2.1	Compilando	9
3.2.2	Executando	9
3.3	Funcionamento	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.

1.1 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

1.1.2 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.

1.1.3 *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()*.

1.1.4 *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]*).

```
int pipe_fd[2]; // Pipe para comunicação
if (pipe(pipe_fd) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
```

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

2.2 *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.

```
pid_t pid = fork();
if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}
```

2.2.1 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.

```
if (pid == 0) { // Subprocesso
    close(pipe_fd[1]); // Fecha a extremidade de escrita no subprocesso
    banking_subprocess(pipe_fd[0], pipe_fd[1]);
    exit(EXIT_SUCCESS);
}
```

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*.

```
} else { // Processo principal
    close(pipe_fd[0]); // Fecha a extremidade de leitura no processo principal

    char command[256];
    while (1) {
        printf("\nDigite o comando seguido pelo valor (depositar, sacar, saldo, sair): ");
        fgets(command, sizeof(command), stdin);
        command[strcspn(command, "\n")] = '\0';

        if (strcmp(command, "sair") == 0) {
            write(pipe_fd[1], "sair", 5);
            break;
        }
        write(pipe_fd[1], command, strlen(command) + 1);
    }

    close(pipe_fd[1]); // Fecha a extremidade de escrita
    wait(NULL); // Aguarda pelo subprocesso
}
```

2.3 Função do Subprocesso:

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

```
void banking_subprocess(int read_fd, int write_fd) {
    char buffer[256];

    while (1) {
        ssize_t bytes_read = read(read_fd, buffer, sizeof(buffer));
        if (bytes_read <= 0) {
            break; // Pipe fechado ou erro
        }

        buffer[bytes_read] = '\0';
        if (strcmp(buffer, "sair") == 0) {
            printf("Saindo do subprocesso bancário...\n");
            break;
        }

        pthread_t thread;
        char *request = strdup(buffer); // Duplicar a string de requisição para a thread
        if (pthread_create(&thread, NULL, handle_request, request) != 0) {
            perror("pthread_create");
            free(request);
        }
        pthread_detach(thread); // Recolher automaticamente os recursos da thread
    }
}
```

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

2.4 Execução das *Threads*:

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

```
void *handle_request(void *arg) {
    char *request = (char *)arg;
    pthread_mutex_lock(&balance_mutex);

    if (strncmp(request, "depositar ", 10) == 0) {
        double amount = atof(request + 10);
        account_balance += amount;
        printf("Depositado %.2f, Novo Saldo: %.2f\n", amount, account_balance);
    } else if (strncmp(request, "sacar ", 6) == 0) {
        double amount = atof(request + 6);
        if (amount > account_balance) {
            printf("Fundos insuficientes! Saldo Atual: %.2f\n", account_balance);
        } else {
            account_balance -= amount;
            printf("Sacado %.2f, Novo Saldo: %.2f\n", amount, account_balance);
        }
    } else if (strcmp(request, "saldo") == 0) {
        printf("Saldo Atual: %.2f\n", account_balance);
    } else {
        printf("Comando desconhecido: %s\n", request);
    }

    pthread_mutex_unlock(&balance_mutex);
    free(request); // Liberar a memória alocada dinamicamente
    return NULL;
}
```

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)*).

3 EXECUÇÃO

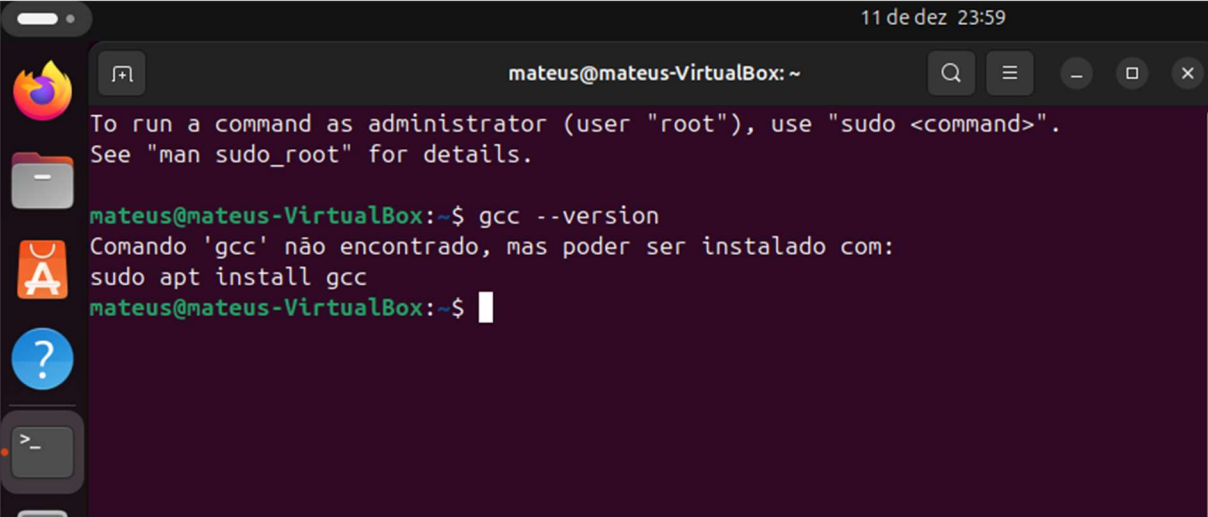
3.1 Preparando o Ambiente

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`:

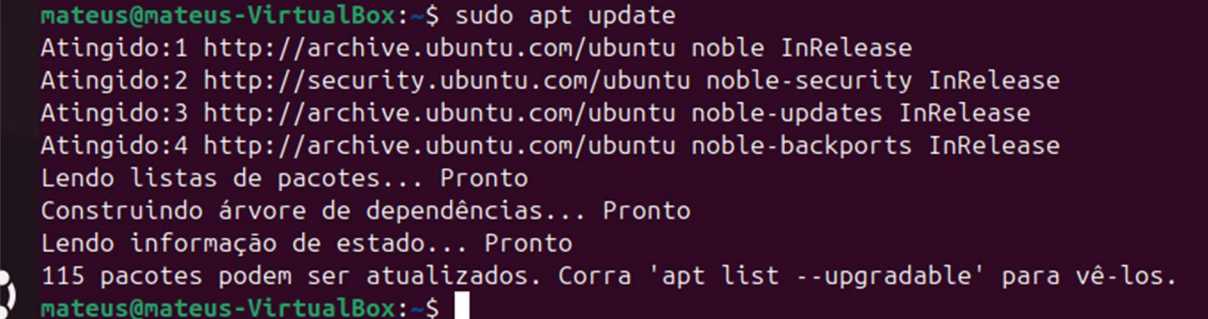


```
mateus@mateus-VirtualBox: ~  
To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo_root" for details.  
mateus@mateus-VirtualBox:~$ gcc --version  
Comando 'gcc' não encontrado, mas poder ser instalado com:  
sudo apt install gcc  
mateus@mateus-VirtualBox:~$
```

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

`sudo apt update`

`sudo apt install gcc`



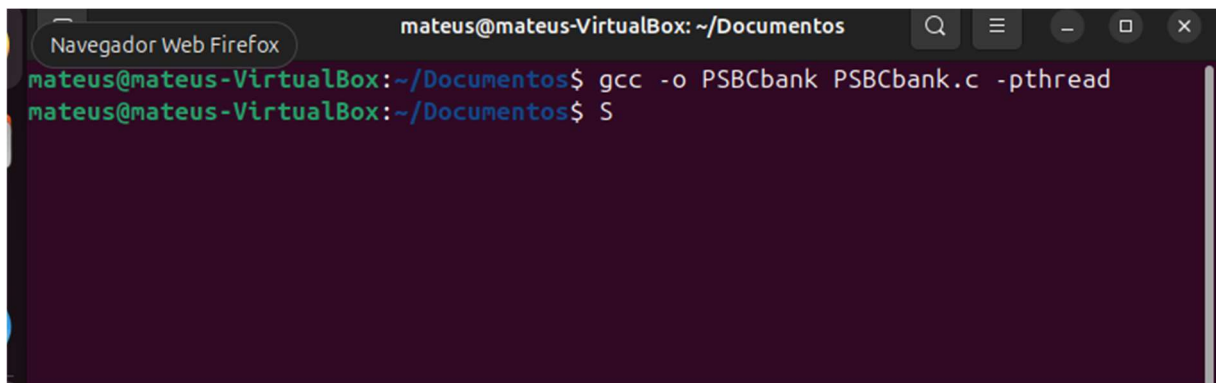
```
mateus@mateus-VirtualBox:~$ sudo apt update  
Atingido:1 http://archive.ubuntu.com/ubuntu noble InRelease  
Atingido:2 http://security.ubuntu.com/ubuntu noble-security InRelease  
Atingido:3 http://archive.ubuntu.com/ubuntu noble-updates InRelease  
Atingido:4 http://archive.ubuntu.com/ubuntu noble-backports InRelease  
Lendo listas de pacotes... Pronto  
Construindo árvore de dependências... Pronto  
Lendo informação de estado... Pronto  
115 pacotes podem ser atualizados. Corra 'apt list --upgradable' para vê-los.  
mateus@mateus-VirtualBox:~$
```

3.2 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)

3.2.1 Compilando:

gcc -o PSBCbank PSBCbank.c -pthread

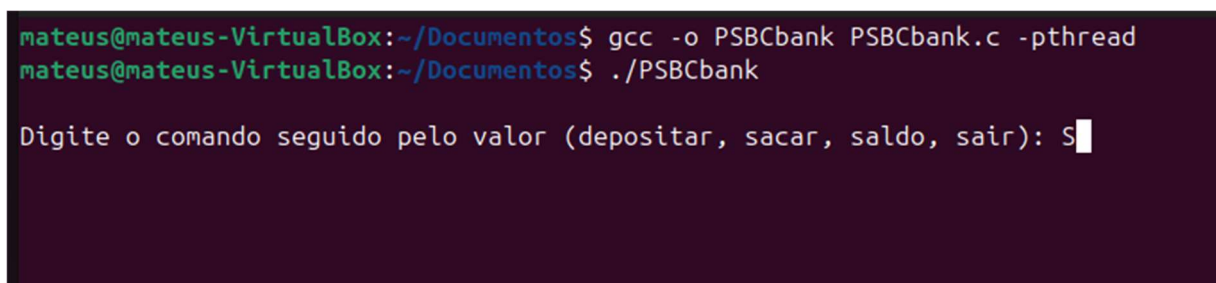
A screenshot of a terminal window titled 'mateus@mateus-VirtualBox: ~/Documentos'. The terminal shows the command 'gcc -o PSBCbank PSBCbank.c -pthread' being entered and executed. The prompt changes to 'mateus@mateus-VirtualBox: ~/Documentos\$' after the command is run. A Firefox browser window is visible in the background.

```
mateus@mateus-VirtualBox: ~/Documentos$ gcc -o PSBCbank PSBCbank.c -pthread
mateus@mateus-VirtualBox: ~/Documentos$
```

3.2.2 Executando:

Para Executar digite:

./PSBCbank

A screenshot of a terminal window showing the execution of the program. The command './PSBCbank' is entered and executed. The prompt changes to 'mateus@mateus-VirtualBox: ~/Documentos\$'. Below the prompt, a message is displayed: 'Digite o comando seguido pelo valor (depositar, sacar, saldo, sair):'. The letter 'S' is entered at the end of the line.

```
mateus@mateus-VirtualBox: ~/Documentos$ gcc -o PSBCbank PSBCbank.c -pthread
mateus@mateus-VirtualBox: ~/Documentos$ ./PSBCbank

Digite o comando seguido pelo valor (depositar, sacar, saldo, sair): S
```

3.3 Funcionamento

```
mateus@mateus-VirtualBox:~/Documentos$ gcc -o PSBCbank PSBCbank.c -pthread
^[[Amateus@mateus-VirtualBox:~/Documentos$ ./PSBCbank

Digite o comando seguido pelo valor (depositar, sacar, saldo, sair): saldo
Digite o comando seguido pelo valor (depositar, sacar, saldo, sair): Saldo Atual: 1000.00
sacar 500

Digite o comando seguido pelo valor (depositar, sacar, saldo, sair): Sacado 500.00, Novo Saldo: 500.00
depositar 1000

Digite o comando seguido pelo valor (depositar, sacar, saldo, sair): Depositado 1000.00, Novo Saldo: 1500.00
sacar 2000

Digite o comando seguido pelo valor (depositar, sacar, saldo, sair): Fundos insuficientes! Saldo Atual: 1500.00
saldo

Digite o comando seguido pelo valor (depositar, sacar, saldo, sair): Saldo Atual: 1500.00
█
```

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.