Sistema de Streaming com Threads

Ana Caroline da Rocha Braz[1]

[1] Departamente de Ciência da Computação (CIC) - Universidade de Brasília (UNB)
Campus Darcy Ribeiro – Asa Norte – Brasília – DF – Brasil – CEP 70910-900

{212008482}@aluno.unb.br

Compilação do Código

Para compilação do código faça: "gcc streming.c - streaming -lpthread"

1. Introdução

No contexto atual da computação, onde a demanda por processamento rápido e eficiente de dados é cada vez mais premente, a programação concorrente emerge como uma abordagem crucial. Lidando com a execução simultânea de múltiplas tarefas ou processos, ela permite que sistemas computacionais realizem operações de maneira mais eficaz, aproveitando recursos e reduzindo tempo de resposta [Alchieri 2022].

A interconexão global e a crescente complexidade das aplicações digitais tornam a programação concorrente essencial. Em sistemas operacionais, a capacidade de gerenciar simultaneamente diversas operações garante a responsividade e eficiência dos dispositivos. Em ambientes web, ela possibilita a manipulação de múltiplas requisições de usuários, assegurando uma experiência fluída e sem interrupções. Além disso, em contextos como jogo multiplayer online e serviços streming, a programação concorrente desempenha um papel crucial para sincronizar ações entre jogadores, garantir uma transmissão contínua de conteúdo e criar experiências de usuário envolventes.

Em razão disto, este trabalho tem por objetivo o desenvolvimento de um algoritmo para tratar a comunicação entre processos através de uma memória compartilhada de uma streaming. Para isso, foi criado um novo problema no qual temos usuários, telas e sincronização audio e vídeo, que precisam trabalhar de forma simultânea e eficiência. Com esse fim, foi implementado um algoritmo que ajude o cliente a ter uma melhor experiência nesse tipo de streaming utilizando mecanismos de sincronização de processos e thread.

Este relatório este dividido da seguinte forma. Na seção 2, estará a formalização do problema proposto. Na Seção 3, está a descrição do algoritmo desenvolvido para solução do problema proposto e, por fim, a conclusão do relatório.

2. Formalização do Problema Proposto

Como dito anteriormente, a programação concorrente desempenha um papel fundamental nos serviços de streaming para a sincronização de ações entre as transmissões de conteúdo e criação de experiência dos usuários envolventes. Dessa forma, foi criado o seguinte problema:

"Considere um sistema de streaming multimídia que oferece serviços de vídeo e áudio para usuários simultaneamente. O sistema deve ser capaz de lidar com múltiplos usuários, proporcionando uma experiência de usuário contínua. Utilizando a biblioteca *POSIX Pthreads* implemente threads separadas para cada usuário, representando suas

sessões individuais de streaming e certifique que o sistema gerencie concorrentemente a transmissão de vídeo e áudio para cada usuário, evitando atrasos perceptíveis entre os dois tipos de mídia. Além disso, considere que existam 5 usuários que gostariam de utilizar o sistema porém apenas 2 sessões podem ser utilizados por vez e, também, a cada 10 interações com o sistema 1 administrador precisará fazer a atualização, precisando de acesso exclusivo ao sistema de streaming."

3. Descrição do Algoritmo Desenvolvido para Solução do Problema Proposto

Para construção do código foram usadas como base o problema dos macacos, problema dos produtor e consumir e problemas e o problema do pombo correio, em que utilizavam semáforos, mutex e váriaveis de condições.

Seguinte a lógica, foi utilizadas as seguintes bibliotecas, foi possível utilizar mutexes, semáforos e barreiras para a construção da implementação.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>
#include <semaphore.h>
```

Listing 1. Bibliotecas utilizadas

A seguir, foi definido o número de usuários (MAX USUARIOS) e o número de telas (MAX SESSOES), todos indicados na questão formalizada.

```
#define MAX_USUARIOS 5
#define MAX_SESSOES 2
```

Listing 2. Definições

Também foram feitas as inicializações das threads da segunte forma:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t turno = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
sem_t tela;
sem_t admin;
int sessoes_em_uso = 0;
int atualizacao = 0;
```

Listing 3. Threads

- pthread_mutex_t mutex: Essa linha declara e inicializa um mutex. O mutex é uma variável usada para garantir que apenas uma thread tenha acesso a uma determinada região crítica do código por vez.
- pthread_mutex_t turno: Semelhante ao primeiro, esta linha declara e inicializa outro mutex chamado turno.
- pthread_cond_t cond : Esta linha declara e inicializa uma variável de condição. Variáveis de condição são frequentemente usadas para sincronizar o acesso a recursos compartilhados entre threads.
- sem_t tela: Essa linha declara uma variável semáforo chamada tela. Semáforos são usados para controlar o acesso concorrente a um recurso, permitindo que várias threads acessem ou limitem o acesso conforme necessário.

- sem_t admin;: Similar à variável tela, esta linha declara um semáforo chamado admin.
- int sessoes_em_uso = 0: Esta variável é utilizada para rastrear quantas sessões estão em uso, possibilitando um possível bloqueo caso mais de 2 telas sejam usadas ao mesmo tempo.
- int atualizacao: flag para saber quando será necessário a atualização do sistema pelo administrador.

A função “thread_usuario” fornece a primeira parte do que foi pedido na questão. Ela faz a com que apenas 2 dos 5 usuários utilizem as sessões disponíveis por vez. É verificado a cada intereção se o máximo de sessões já foi atingido, em caso positivo a thread entra em modo de espera usando a pthread_cond_wait e só é liberada quando outra thread sinaliza que existe sessão disponível. Também é verificada a quantidade de intereções por meio da atualização, assim que chega a 10 é liberado o semáforo ao administrador para fazer a atualização do sistema.

```
void *thread_usuario(void *arg) {
    int id_usuario = *((int *)arg);

    while (1) {
        // Inicia sess o do usu rio
        sem_wait(&tela);
        pthread_mutex_lock(&mutex);

        while (sessoes_em_uso >= MAX_SESSOES) {
            // Aguarda disponibilidade de sess o
            pthread_cond_wait(&cond, &mutex);
        }

        sessoes_em_uso++;
        pthread_mutex_unlock(&mutex);

            // Realiza streaming de v deo e  udio  simultaneamente
            printf(">> Usuario %d esta fazendo streaming de video e
    audio... << \n", id_usuario);
            sleep(2);  // Simula o streaming
            atualizacao++;

        // Encerra sess o do usu rio
        pthread_mutex_lock(&mutex);
        sessoes_em_uso--;
        printf("\nAtualizacao daqui a %d streaming\n", atualizacao);
        pthread_cond_signal(&cond);  // Libera uma sess o para outros
    usu rios
        printf("<< Streaming  do usuartio %d finalizada >>\n\n",
    id_usuario);
        if(atualizacao == 10){
            sem_post(&admin);
        }
        pthread_mutex_unlock(&mutex);
        sem_post(&tela);
    }
     pthread_exit(0);
```

```
}
```

Listing 4. Função do Usuário

A função “thread_administrador” fornece a segunda parte do que foi pedido na questão. Com exclusão mútua, o administrador pode fazer a atualização do sistema sem que seja interrompido.

```
void *thread_administrador(void *arg) {
    while (1) {
        // Realiza opera  es administrativas exclusivas
        sem_wait(&admin);
        pthread_mutex_lock(&mutex);

            printf("\nO administrador esta realizando a atualizacao do sistema...\n");
            sleep(5);
            printf("Atualizacao do sistema realizada com sucesso!\n");
            atualizacao = 0;

        pthread_mutex_unlock(&mutex);
        sem_post(&tela);
    }
     pthread_exit(0);
}
```

Listing 5. Função do Administrador

Por fim, temos a main do código, onde é feita as inicializações e criações das threads.

```
int main() {
    pthread_t tid_administrador;
    pthread_t tid_usuario[MAX_USUARIOS];
    int ids_usuarios[MAX_USUARIOS];

    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&turno, NULL);

    sem_init(&tela, 0, MAX_SESSOES);
    sem_init(&admin, 0, 0);

    // Inicia a thread do administrador
    pthread_create(&tid_administrador, NULL, thread_administrador, NULL);

    // Inicia as threads dos usu rios
    for (int i = 0; i < MAX_USUARIOS; i++) {
        ids_usuarios[i] = i + 1;
        pthread_create(&tid_usuario[i], NULL, thread_usuario, &ids_usuarios[i]);
    }

    // Aguarda as threads terminarem (nunca deve acontecer no exemplo)
    pthread_join(tid_administrador, NULL);
    for (int i = 0; i < MAX_USUARIOS; i++) {
        pthread_join(tid_usuario[i], NULL);
```

```
25      }
26      return 0;
27      }
```

Listing 6. Função do Main

3.1. Resultados

Ao compilar e executar o código feito chegamos ao seguinte resultado (Figura 1):

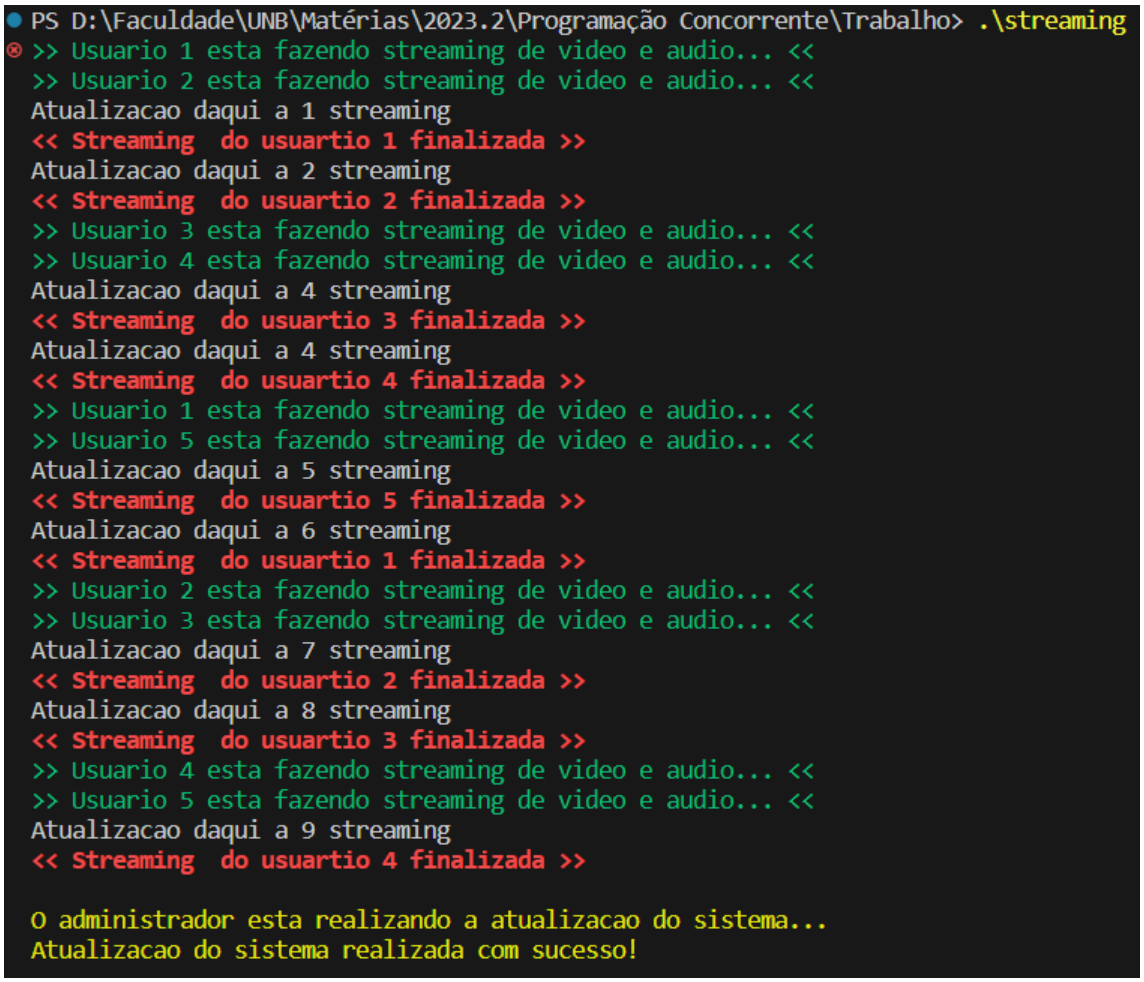

Figura 1. Resultado

Pode-se obeservar que apenas 2 sessões são utilizadas por vez e que a contagem da atualização é feita de acordo com as sessões utilizadas. Sendo assim, quando chegar a 10 o administrador tem acesso exclusivo e consegue fazer a atualização do sistema.

4. Conclusão

O seguinte trabalho apresentou sobre o que é programação concorrente, suas utilidades e como ela está inserida no meio de sistemas de streaming. Para mostrar na prática, foi formalizado uma questão onde há 5 usuários querendo utilizar um sistema de streaming, no entanto apenas 2 sessões podiam ser utilizadas ao mesmo tempo e, quando chegasse a um certo número de streamings, um administrador precisaria realizar a atualização do

sistema. Para isso foi implementado um código utilizando a biblioteca *POSIX Pthreads* e executado mostrando como isso aconteceria, chegando ao resultado final esperado.

Referências

Alchieri, E. (2022). Slides 1 - programação concorrente.