Relatório - Barreira e Fila FIFO Para Processos

Luan Matheus Trindade Dalmazo
Mateus de Oliveira Silva

Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil

I. Introducão

Este relatório tem por objetivo detalhar os aspectos da implementação empregados no trabalho ”Barreira e Fila FIFO Para Processos”, bem como, fornecer instruções quanto ao comando para rodar a aplicação.

II. O código

O projeto segue a estrutura evidenciada na Figura 1. A lógica de cada diretório é detalhada a seguir:

- src: Contém os códigos-fonte.
- include: Contém os arquivos de cabeçalho.
- obj: Irá conter os arquivos gerados pela compilação.
- bin: Irá conter o executável binário gerado pela *linkagem*.

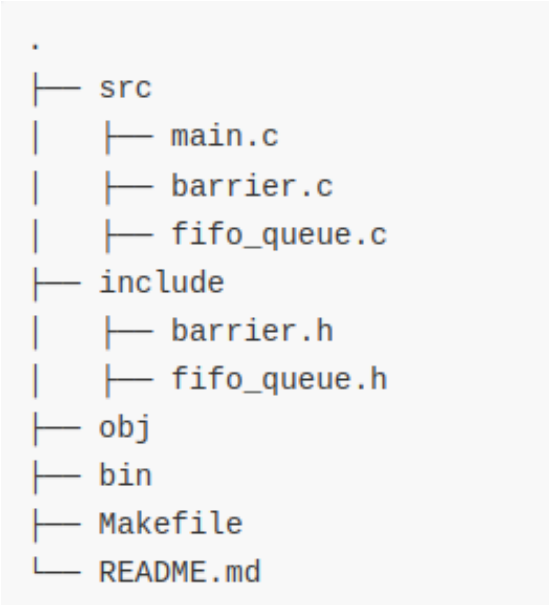

Figura 1. Estrutura da aplicação.

O arquivo *README.md* fornece orientações gerais para uso da aplicação.

A. Executando a aplicação

Para rodar a aplicação basta rodar o comando retratado abaixo:

```
make run
```

Em seguida, será solicitado o número de processos que o usuário deseja realizar o teste, por padrão, a aplicação espera um valor maior que 0.

III. Barreira

Para implementação da barreira, é feito uso de duas funções: init_barrier III-A e process_barrier III-B que operam na estrutura visualizada abaixo:

```
typedef struct barrier_s {
    int total_processes;
    int waiting_processes;
    sem_t mutex;
    sem_t barrier_semaphore;
} barrier_t;
```

Duas estruturas de semáforos são utilizadas, *mutex* para controle de acesso a barreira e *barrier_semaphore* para gerenciamento dos processos na estrutura, de modo a garantir o bloqueio até que todos processos tenham chegado à barreira.

A. init_barrier(barrier_t *barr, int n)

Para inicializar a barreira, essa função recebe um ponteiro para uma estrutura do tipo *barrier_t* e um inteiro n, que representa o número de *threads* que irão utilizar a barreira. Após isso, *total_processes* recebe o valor *n* e *waiting_processes* recebe o valor 0, representando o início da barreira.

Os dois semáforos utilizados são iniciados, *mutex* com o valor 1 representando o estado de desbloqueado e *barrier_semaphore* com o valor 0, representando o estado de bloqueado.

B. process_barrier(barrier_t *barr)

A função *process_barrier* atua sobre a barreira, de forma a realizar a sincronização dos processos. De início, o *mutex* é bloqueado, e o número de processos que estão em espera é incrementado em 1. Após isso, é feita uma verificação se todos os processos chegaram na barreira, se for o caso, o semáforo da barreira é desbloqueado, e o número de processos que estão aguardando é reduzido para 0. No final, o *mutex* é desbloqueado e o semáforo da barreira é bloqueado novamente.

IV. Fila FIFO

Para implementação da fila FIFO, são utilizadas três funções: `init_fifoQ` IV-A, `espera` IV-B `liberaPrimeiro` IV-C que operam na estrutura visualizada abaixo:

```
typedef struct fifoQ_s {
    sem_t mutex;
    sem_t wait_sem;
    int waiting_count;
} FifoQT;
```

Duas estruturas de semáforos são utilizadas: `mutex` para controle de acesso à fila e `wait_sem` para gerenciamento dos processos na estrutura, de modo a garantir que os processos aguardem em ordem até serem liberados para avançar.

A. A. *init_fifoQ(FifoQT *F)*

Para inicializar a fila FIFO, essa função recebe um ponteiro para uma estrutura do tipo `FifoQT`. Inicialmente, o semáforo `mutex` é definido com o valor 1, representando o estado de desbloqueado, enquanto `wait_sem` é iniciado com o valor 1, representando também o estado de desbloqueado. A variável `waiting_count` recebe o valor 0, indicando que não há processos em espera no início.

B. B. *espera(FifoQT *F)*

A função `espera` gerencia a sincronização dos processos na fila. Inicialmente, o semáforo `mutex` é bloqueado, e o número de processos em espera, `waiting_count`, é incrementado em 1. Em seguida, o processo entra em estado de espera ao bloquear-se no semáforo `wait_sem`. Após isso, o semáforo `mutex` é liberado, permitindo que outros processos possam se inserir na fila.

C. C. *liberaPrimeiro(FifoQT *F)*

A função `liberaPrimeiro` atua sobre a fila, de forma a liberar o primeiro processo em espera. Primeiramente, o semáforo `mutex` é bloqueado, e a função verifica se há processos aguardando, decrementando o valor de `waiting_count`. Em seguida, o semáforo `wait_sem` é desbloqueado, permitindo que o processo na frente da fila continue a execução. Ao final, o semáforo `mutex` é desbloqueado, permitindo a próxima operação na fila.

V. Correção do Bug de Sincronização na Fila FIFO com Contagem de Processos em Espera

Durante a implementação do trabalho, surgiu um `bug` na fila FIFO que forneceu um "insight" importante. Esse `bug` permitia que dois ou mais processos acessassem a área de uso exclusivo simultaneamente, o que não deveria ocorrer. Após várias tentativas para entender o problema, identificou-se que o erro estava na função de espera, que originalmente estava implementada da seguinte forma:

```
void espera(FifoQT *F) {
    lock_queue_struct(F);
    if(is_empty(F)) {
        unlock_queue_struct(F);
        return;
    }

    F->waiting_count++;
    unlock_queue_struct(F);
    wait_in_queue(F);
}
```

O problema era que, quando a fila estava vazia, o processo passava direto e não ficava na espera, sem incrementar o `waiting_count`. Inicialmente, isso parecia fazer sentido, mas, ao analisar mais profundamente, percebemos que não era o comportamento esperado.

Para ilustrar o problema, utilizamos a analogia de um consultório médico: o prólogo corresponde à sala de espera, o uso representa a sala de consulta, e o epílogo simboliza a saída. O que estava acontecendo era similar a um processo que, ao entrar na sala de espera e encontrá-la vazia, segue diretamente para a sala de consulta, sem considerar que outro processo já poderia estar sendo atendido. Ao ignorar a presença de um processo em consulta e tratar a fila como vazia, gerava-se uma situação em que um recurso de uso exclusivo era acessado simultaneamente por vários processos.

A solução para esse problema foi adicionar o incremento de `F->waiting_count++` dentro da verificação de fila vazia, conforme o código atualizado abaixo:

```
if(is_empty(F)) {
    F->waiting_count++;
    unlock_queue_struct(F);
    return;
}
```

Essa modificação garante que nenhum processo entre na área de uso exclusivo enquanto outro processo já estiver presente.

VI. Exemplo Prático de Uso

Nesta seção, será abordado um exemplo prático com a utilização da aplicação para o caso de 3 processos filhos (n = 3). Para melhor exemplificação, foram impressos o tempos dos processos e também algumas breves descrições para indicar o início do processo. Na Figura 2 é possível visualizar a execução do exemplo prático.

O fluxo da aplicação consiste de um conjunto de processos que são sincronizados na barreira, e após isso, os processos são encaminhados a um *loop* de usos administrados pela fila FIFO. Durante o *loop* de uso, três simulações são executadas:

- (A) Prologo
- (B) Uso com exclusividade
- (C) Epilogo

Os tempos foram evidenciados na figura para demonstrar que durante o caso B, apenas um processo pode estar rodando o uso, então para que um outro processo adentre essa etapa, deve aguardar a finalização do processo.

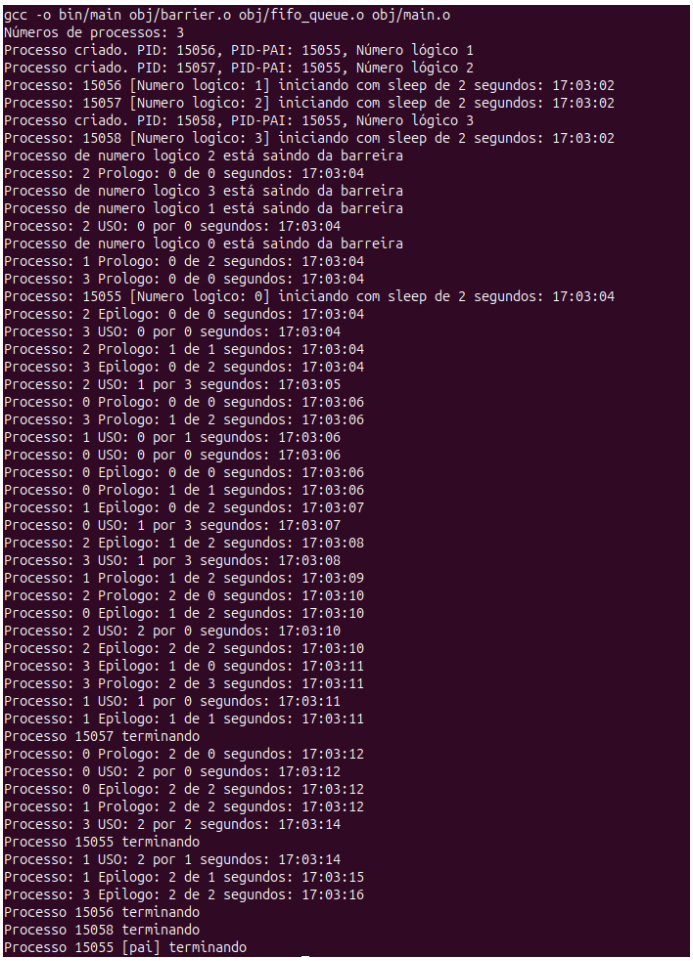

Figura 2. Resultado para o exemplo prático.