

# 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. INTRODUÇÃO

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*.

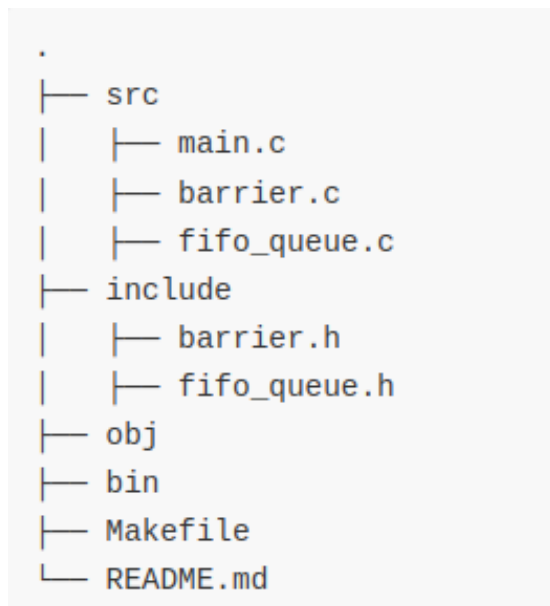


Figura 1. Estrutura da aplicação.

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, int nProc)*

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, o semáforo da barreira é bloqueado novamente e o número lógico (*nProc*) do processo que está saindo da barreira é impresso.

#### IV. FILA FIFO

Para implementação da fila FIFO, são utilizadas três funções: `init_fifoQ` **IV-A**, `espera` **IV-B** e `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.

```

gcc -o bin/main obj/barrier.o obj/fifo_queue.o obj/main.o
Números de processos: 3
Processo criado. PID: 15056, PID-PAI: 15055, Número lógico 1
Processo criado. PID: 15057, PID-PAI: 15055, Número lógico 2
Processo: 15056 [Número lógico: 1] iniciando com sleep de 2 segundos: 17:03:02
Processo: 15057 [Número lógico: 2] iniciando com sleep de 2 segundos: 17:03:02
Processo criado. PID: 15058, PID-PAI: 15055, Número lógico 3
Processo: 15058 [Número lógico: 3] iniciando com sleep de 2 segundos: 17:03:02
Processo de numero logico 2 está saindo da barreira
Processo: 2 Prologo: 0 de 0 segundos: 17:03:04
Processo de numero logico 3 está saindo da barreira
Processo de numero logico 1 está saindo da barreira
Processo: 2 USO: 0 por 0 segundos: 17:03:04
Processo de numero logico 0 está saindo da barreira
Processo: 1 Prologo: 0 de 2 segundos: 17:03:04
Processo: 3 Prologo: 0 de 0 segundos: 17:03:04
Processo: 15055 [Número lógico: 0] iniciando com sleep de 2 segundos: 17:03:04
Processo: 2 Epilogo: 0 de 0 segundos: 17:03:04
Processo: 3 USO: 0 por 0 segundos: 17:03:04
Processo: 2 Prologo: 1 de 1 segundos: 17:03:04
Processo: 3 Epilogo: 0 de 2 segundos: 17:03:04
Processo: 2 USO: 1 por 3 segundos: 17:03:05
Processo: 0 Prologo: 0 de 0 segundos: 17:03:06
Processo: 3 Prologo: 1 de 2 segundos: 17:03:06
Processo: 1 USO: 0 por 1 segundos: 17:03:06
Processo: 0 USO: 0 por 0 segundos: 17:03:06
Processo: 0 Epilogo: 0 de 0 segundos: 17:03:06
Processo: 0 Prologo: 1 de 1 segundos: 17:03:06
Processo: 1 Epilogo: 0 de 2 segundos: 17:03:07
Processo: 0 USO: 1 por 3 segundos: 17:03:07
Processo: 2 Epilogo: 1 de 2 segundos: 17:03:08
Processo: 3 USO: 1 por 3 segundos: 17:03:08
Processo: 1 Prologo: 1 de 2 segundos: 17:03:09
Processo: 2 Prologo: 2 de 0 segundos: 17:03:10
Processo: 0 Epilogo: 1 de 2 segundos: 17:03:10
Processo: 2 USO: 2 por 0 segundos: 17:03:10
Processo: 2 Epilogo: 2 de 2 segundos: 17:03:10
Processo: 3 Epilogo: 1 de 0 segundos: 17:03:11
Processo: 3 Prologo: 2 de 3 segundos: 17:03:11
Processo: 1 USO: 1 por 0 segundos: 17:03:11
Processo: 1 Epilogo: 1 de 1 segundos: 17:03:11
Processo 15057 terminando
Processo: 0 Prologo: 2 de 0 segundos: 17:03:12
Processo: 0 USO: 2 por 0 segundos: 17:03:12
Processo: 0 Epilogo: 2 de 2 segundos: 17:03:12
Processo: 1 Prologo: 2 de 2 segundos: 17:03:12
Processo: 3 USO: 2 por 2 segundos: 17:03:14
Processo 15055 terminando
Processo: 1 USO: 2 por 1 segundos: 17:03:14
Processo: 1 Epilogo: 2 de 1 segundos: 17:03:15
Processo: 3 Epilogo: 2 de 2 segundos: 17:03:16
Processo 15056 terminando
Processo 15058 terminando
Processo 15055 [pai] terminando

```

Figura 2. Resultado para o exemplo prático.