Sincronização com Semáforos e Barreira

Barbara Reis dos Santos e Mayara Lessnau de Figueiredo Neves

CI1215 - Sistemas Operacionais, 1º Semestre de 2025
Prof. Wagner M. Nunan Zola

1 Introdução

Este relatório descreve o desenvolvimento e funcionamento de um programa em C que utiliza processos, semáforos e memória compartilhada para coordenar múltiplas execuções concorrentes. A implementação combina dois mecanismos principais: uma barreira de sincronização, que garante que todos os processos alcancem um ponto comum de execução antes de prosseguir, e um sistema de controle de acesso exclusivo a um recurso compartilhado baseado em fila FIFO (First In, First Out). Dessa forma, o programa assegura tanto a sincronização coletiva quanto o acesso ordenado e justo aos recursos, independentemente da política interna de agendamento do sistema operacional.

2 Objetivo do Código

O objetivo principal é implementar e demonstrar o funcionamento de:

- Uma **barreira de sincronização**, permitindo que um grupo de processos apenas prossiga após todos alcançarem o mesmo ponto do programa. Essa barreira deve ser reutilizável.
- Primitivas de **uso exclusivo e liberação de um recurso compartilhado**, com controle de acesso via uma **fila FIFO**, garantindo que os processos utilizem o recurso em ordem de chegada.
- Uso de **memória compartilhada** entre os processos para armazenar estruturas como a barreira e a fila de espera.
- Um laço de execução em que cada processo utiliza o recurso 3 vezes, com **prólogo**, **uso exclusivo** e **epílogo**, incluindo simulação de tempo de execução via `sleep()`.
- Registro da ordem de término dos processos, exibida **apenas pelo processo pai**.

O código é dividido em duas partes principais (A e B), com a implementação de duas bibliotecas locais: barrier.h e fifo.h. A primeira biblioteca, barrier.h, é responsável pela implementação da barreira de sincronização, enquanto a segunda, fifo.h, gerencia o controle de acesso ao recurso compartilhado por meio de uma fila FIFO (First In First Out). Ambas as bibliotecas utilizam semáforos e memória compartilhada para coordenar os processos, garantindo a sincronização adequada e o acesso ordenado ao recurso compartilhado.

3 Como Executar o Programa

Para compilar e executar o programa, é necessário seguir os seguintes passos:

1. Acessar o diretório do projeto onde se encontra o arquivo `Makefile`.
2. Executar o comando abaixo para compilar o programa:

```
make
```

3. Executar o programa com o seguinte comando, substituindo `<num_processos>` pela quantidade de processos desejada (por exemplo: 5):

```
./trab1 <num_processos>
```

4 Estrutura do Projeto

O desenvolvimento do programa foi estruturado de forma modular, com o objetivo de organizar melhor o código, facilitar a reutilização e leitura de componentes. Para isso, o projeto foi dividido em arquivos de cabeçalho (`.h`) e arquivos de implementação (`.c`), conforme descrito abaixo:

- `fifo.h` e `fifo.c`: responsáveis pela implementação da fila circular com controle de acesso concorrente, utilizada para garantir exclusão mútua no uso do recurso compartilhado entre os processos. Essa fila segue a política **FIFO** (First-In, First-Out), garantindo que os

processos aguardem e acessem o recurso compartilhado exatamente na ordem em que solicitaram o uso.

- `barrier.h` e `barrier.c`: responsáveis pela implementação da barreira de sincronização, que garante que todos os processos iniciem e finalizem os ciclos de execução de forma coordenada.

- `trab1.c`: arquivo principal do programa, onde está localizada a função `main()`. É responsável pela criação dos processos, inicialização das estruturas compartilhadas e controle geral da execução.

5 Descrição das Estruturas

5.1 FIFO (Exclusão Mútua)

A estrutura `FifoQT` implementa uma fila de espera circular para o controle de acesso ao recurso compartilhado. Quando um processo não consegue acessar o recurso imediatamente, ele entra na fila e aguarda sua vez.

Listagem 1: Estrutura FifoQT

```
typedef struct {
    int usando;
    int head;
    int tail;
    int num_procs;
    sem_t fila[MAX_PROCS];
    sem_t lock;
} FifoQT;
```

- `usando`: Indica se o recurso está em uso (1) ou disponível (0).

- `head`: Aponta para o próximo processo a ser liberado.

- `tail`: Aponta para a próxima posição disponível na fila.

- `num_procs`: Armazena a quantidade de processos passada como parâmetro na linha de comando.

- `fila`: Armazena semáforos para cada processo em espera.

- `lock`: Semáforo que protege o acesso à fila.

5.2 Barreira

A estrutura `barrier_t` é usada para sincronizar os processos em pontos determinados da execução, como antes do uso do recurso e após o término.

Listagem 2: Estrutura barrier_t

```
typedef struct shmseg {
    int total;
    int cont;
    sem_t mutex;
    sem_t semaforo;
} barrier_t;
```

- `total`: Número de processos que devem atingir a barreira.

- `cont`: Contador de quantos já chegaram.

- `mutex`: Semáforo para evitar condição de corrida no contador.

- `semaforo`: Controla a liberação dos processos na barreira.

5.3 Dados Compartilhados

A estrutura `shared_data_t` agrupa as estruturas de barreira e a fila FIFO que vão ser compartilhadas pelos processos.

Listagem 3: Estrutura shared_data_t

```
typedef struct {
    FifoQT fila;
    barrier_t barr;
} shared_data_t;
```

- `fila`: Instância da fila FIFO para controle de acesso ao recurso.

- `barr`: Instância da barreira para sincronização dos processos.

6 Funcionamento Geral do Programa

Nesta seção, será detalhada a lógica utilizada para cada uma das funções do programa. Para facilitar o entendimento, vamos separar o código em Inicialização, Execução dos Processos, Detalhamento das Estruturas de Controle e Finalização.

6.1 Inicialização

A inicialização acontece principalmente na função `main` do programa. O processo principal realiza os seguintes passos:

- Cria uma área de memória compartilhada com `shmget` e associa essa área a um ponteiro com `shmat`.

- Inicializa a estrutura da fila com `init_fifoQ`, definindo que o recurso está livre e os semáforos da fila estão bloqueados.

- Inicializa a barreira com `init_barr`, definindo o número de processos que devem se sincronizar. O semáforo da barreira começa fechado.
- Inicializa o semáforo `lock_termino` com valor 1, garantindo exclusão mútua para registrar a ordem de término dos processos. O semáforo começa livre: inicialmente, todos podem acessar a memória compartilhada.
- Cria os processos filhos com `fork()`, armazenando os PID de cada filho em um vetor para controle posterior.

6.2 Execução dos Processos

Cada processo executa a função `ciclo_cliente`, que realiza os seguintes passos:

1. **Barreira Inicial:** Todos os processos esperam na barreira até que todos estejam prontos para iniciar. Isso é feito com a chamada da função `process_barrier`, que libera os processos apenas quando todos atingirem a barreira.
2. **Ciclos de Uso do Recurso:** Cada processo realiza três ciclos com as seguintes etapas:
   - **Prólogo:** Simula uma preparação com um atraso aleatório.
   - **Uso (Região Crítica):** O processo entra na fila para acessar o recurso de forma exclusiva:
     - Se o recurso está livre e a fila está vazia, o processo passa direto e inicia o uso.
     - Caso contrário, entra na fila e espera no seu semáforo.
     - O uso é simulado com um `sleep()`.
     - Ao liberar o recurso, o processo acorda o próximo da fila ou, se a fila estiver vazia, apenas libera o recurso.
   - **Epílogo:** Simula o encerramento da atividade com outro atraso aleatório.
3. **Barreira Final:** Após os três ciclos, os processos esperam novamente até que todos tenham terminado seus ciclos antes de prosseguir (segunda chamada da função `process_barrier`).

6.3 Detalhamento das Estruturas de Controle

Barreira

A barreira faz com que os processos esperem todos chegarem para começar o ciclo e, depois, que todos esperem todos terminarem para sair do ciclo. Ela é baseada em duas funções:

- `init_barr`
  - O campo `total` é inicializado com `n`, que representa o número de processos e é passado como parâmetro na linha de comando ao executar o código.
  - O `contador` é inicializado em 0.
  - O mutex de controle de acesso ao contador é inicializado destravado com `sem_init`.
  - O semáforo que controla se os processos devem esperar ou sair da barreira também é inicializado com `sem_init` e começa bloqueado.
- `process_barrier`
  - Cada processo, ao chegar à função, incrementa a variável contadora: `cont++`.
  - O acesso à variável `cont` é protegido com um mutex. Antes de ser incrementada, o mutex é travado com `sem_wait` e só é destravado com `sem_post` após a modificação, garantindo exclusão mútua.
  - O último processo a chegar (quando `cont == total`) libera os demais.
  - Ao final, `cont` é resetada para 0, permitindo que a barreira seja reutilizada sem necessidade de nova chamada à `init_barr`.

Fila FIFO com Exclusão Mútua

A fila controla o acesso exclusivo ao recurso e é baseada em três funções:

- `init_fifoQ`:
  - Inicializa a fila com o recurso livre, ou seja, `usando = 0`.
  - `head` e `tail` começam na posição 0 do vetor circular.
  - Inicializa os semáforos do vetor com `sem_init` em um laço `for`.
  - Inicializa o semáforo que controla o acesso à fila com `sem_init`.
  - Define a quantidade total de processos que irão interagir com a fila, atribuindo o valor ao campo `num_procs`.
- `inicia_uso`:
  - Se o recurso está livre e a fila está vazia, o processo passa direto, sem entrar na fila, e o recurso passa a estar sendo usado (`usando = 1`).
  - Caso contrário, o processo entra na fila e espera seu semáforo com `sem_wait()`.

- `termina_uso`:
  - Se a fila estiver vazia, o processo marca o recurso como livre: `usando = 0`.
  - Caso contrário, acorda o próximo processo da fila com `sem_post()` na posição `head`, ou seja, o primeiro da fila.

6.4 Finalização

Ao término da execução, o processo pai aguarda a finalização de todos os processos filhos, imprimindo a ordem em que cada um encerrou sua execução com base nos retornos da chamada `wait`. Após garantir que todos os processos concluíram corretamente, o pai realiza a limpeza dos recursos utilizados: todos os semáforos são destruídos e a memória compartilhada é devidamente liberada por meio da função `shmctl`.

7 Ordem de Término dos Processos

A ordem de término dos processos no programa é determinada pela sequência em que o processo pai recebe os sinais de finalização dos processos filhos. Esse controle é feito por meio da função `wait()`, chamada em um laço logo após a execução do processo pai na função `main()`.

O funcionamento é descrito em detalhes a seguir:

- Cada processo filho é criado a partir da chamada `fork()`, dentro de um laço que vai de 1 até `num_procs - 1`. O processo pai mantém um vetor `pids[]` onde armazena o `PID` real retornado por cada `fork()`, associando-o ao número lógico do processo correspondente.
- Após a criação dos filhos, o processo pai também executa a função `ciclo_cliente()` como processo lógico 0.
- Finalizada sua execução, o processo pai entra em um laço onde chama `wait()` repetidamente. A função `wait()` bloqueia a execução até que um dos processos filhos termine, retornando seu `PID`.
- O pai percorre o vetor `pids[]` para identificar a qual número lógico pertence o `PID` retornado.
- Dessa forma, a ordem das mensagens exibidas representa a sequência em que os processos filhos finalizaram sua execução, do ponto de vista do sistema operacional.
- Essa ordem não necessariamente reflete a ordem lógica dos processos nem a ordem de criação, mas sim a ordem de término real observada pelo processo pai, à medida que o sistema notifica o fim de cada filho.

Essa estratégia é eficiente para registrar e exibir a ordem de término dos processos de forma segura, utilizando apenas chamadas de sistema e recursos do próprio processo pai.

8 Resultados

Para validar o funcionamento correto da sincronização por semáforos e barreiras, realizamos diversos testes com diferentes quantidades de processos. A seguir, apresentamos como exemplo a execução com quatro processos — ou seja, rodamos o programa com o comando `./trab1 4`. Os trechos a seguir estão divididos por etapas e explicam o comportamento observado na saída do código.

8.1 Entrada na Primeira Barreira

Logo no início, todos os processos chegam na primeira barreira e aguardam até que todos tenham chegado:

```
--Processo: 1 chegando na barreira
--Processo: 2 chegando na barreira
--Processo: 3 chegando na barreira
--Processo: 0 chegando na barreira
```

Esse comportamento confirma que a barreira está funcionando corretamente: nenhum processo deve prosseguir até que todos tenham alcançado esse ponto.

8.2 Saída da Primeira Barreira e Início da Região Crítica

Depois que todos chegaram, os processos começam a sair da barreira (marcados com ‘**‘) e iniciam sua execução normal, que consiste em ciclos de **prólogo**, **uso** do recurso (região crítica) e **epílogo**.

```
**Processo: 0 saindo da barreira
Processo: 0 Prologo: 0 de 3 segundos
**Processo: 3 saindo da barreira
**Processo: 2 saindo da barreira
Processo: 3 Prologo: 0 de 2 segundos
Processo: 2 Prologo: 0 de 0 segundos
**Processo: 1 saindo da barreira
Processo: 1 Prologo: 0 de 2 segundos
```

Todos os processos iniciam o prólogo — uma fase simulada com `sleep()` — e se preparam para entrar na região crítica.

8.3 Acesso Exclusivo ao Recurso (Região Crítica)

Durante o uso do recurso, somente um processo por vez entra na região crítica, como mostrado abaixo:

```
Processo: 2 USO: 0 por 2 segundos
Processo: 2 Epilogo: 0 de 0 segundos
Processo: 1 USO: 0 por 2 segundos
Processo: 1 Epilogo: 0 de 3 segundos
Processo: 3 USO: 0 por 3 segundos
Processo: 3 Epilogo: 0 de 3 segundos
Processo: 0 USO: 0 por 1 segundos
Processo: 0 Epilogo: 0 de 3 segundos
```

Aqui fica claro que os semáforos e a fila circular estão controlando corretamente o acesso ao recurso. Nenhum processo entra na região crítica antes do outro terminar (passar pelo epílogo) e sair dela.

8.4 Repetição dos Ciclos de Execução

Cada processo repete esse ciclo de prólogo, uso e epílogo três vezes (como definido por `MAX_USOS`). Trecho da segunda rodada:

```
Processo: 2 Prologo: 1 de 2 segundos
Processo: 1 Prologo: 1 de 0 segundos
Processo: 3 USO: 1 por 3 segundos
Processo: 0 Prologo: 1 de 1 segundos
...
Processo: 0 USO: 1 por 3 segundos
...
Processo: 2 USO: 2 por 3 segundos
...
Processo: 1 USO: 2 por 2 segundos
...
```

A ordem dos usos pode variar de execução pra execução, mas o mais importante é que continua havendo **exclusão mútua** — só um processo por vez entra no uso.

8.5 Entrada na Segunda Barreira

Após completar os três usos do recurso, os processos alcançam a segunda barreira:

```
--Processo: 2 chegando novamente na
    barreira
--Processo: 1 chegando novamente na
    barreira
--Processo: 3 chegando novamente na
    barreira
--Processo: 0 chegando novamente na
    barreira
```

Assim como na primeira, a segunda barreira impede que qualquer processo finalize sua execução antes que todos tenham completado os três ciclos.

8.6 6. Saída Final e Encerramento dos Processos

Após todos chegarem à barreira, eles saem juntos (marcado com '++') e encerram sua execução. O processo pai imprime mensagens de término dos filhos na ordem em que terminaram:

```
++Processo: 0 saindo da barreira
    novamente
++Processo: 1 saindo da barreira
    novamente
++Processo: 3 saindo da barreira
    novamente
++Processo: 2 saindo da barreira
    novamente
+++ Filho de numero logico 1 e pid 5686
     terminou!
+++ Filho de numero logico 2 e pid 5687
     terminou!
+++ Filho de numero logico 3 e pid 5688
     terminou!
```

Esse trecho final confirma que os processos só terminam juntos, após toda a execução controlada e sincronizada.

8.7 Resumo dos Resultados

A execução confirma que:

- As **barreiras** garantem sincronização total no início e fim da execução.
- A **exclusão mútua** é respeitada: apenas um processo por vez acessa a região crítica.
- A **fila de espera** e o controle por semáforos funcionam corretamente, organizando a ordem de acesso.

9 Discussão: Possível Extensão para Múltiplos Recursos

O programa atual considera a existência de um único recurso compartilhado. Para estender a lógica e suportar múltiplos recursos, seria necessário modificar tanto as estruturas de dados quanto as funções de controle de acesso. As principais adaptações são descritas a seguir:

- **Criação de múltiplas filas:** Substituir a estrutura única `FifoQT` por um vetor de filas, por exemplo, `FifoQT fila[MAX_RECURSOS]`, onde cada elemento gerencia o acesso a um recurso distinto. Cada fila manteria seu próprio conjunto de semáforos e indicadores de uso.
- **Inicialização adaptada:** A função `init_fifoQ` deve ser chamada em um loop para inicializar todas as filas do vetor, garantindo que cada recurso seja tratado de forma independente.

- **Evitar deadlocks:** Caso o programa seja estendido para permitir que um mesmo processo use múltiplos recursos simultaneamente, será necessário implementar uma estratégia de ordenação para aquisição dos recursos, como a política de menor índice primeiro. Essa abordagem evita ciclos de espera circular que resultam em deadlock.
- **Sincronização por recurso:** A sincronização deve permanecer por recurso — ou seja, o semáforo de cada fila controla exclusivamente seu próprio recurso, sem interdependência entre filas.
- **Exemplo de uso com múltiplos recursos:** Um processo poderia, por exemplo, tentar usar o recurso 0, fazer uma pausa, e depois usar o recurso 2. O controle de acesso ocorreria de forma isolada em cada caso:

Listagem 4: Exemplo

```
inicia_uso(0, &fila[0]);
// uso do recurso 0
termina_uso(&fila[0]);

inicia_uso(2, &fila[2]);
// uso do recurso 2
termina_uso(&fila[2]);
```

Essa extensão representa um cenário mais próximo de aplicações reais de sistemas concorrentes, em que diversos recursos disputados coexistem. No entanto, ela introduz maior complexidade no controle de concorrência, exigindo atenção especial para evitar condições de corrida e bloqueios mútuos.

10 Conclusão

Neste trabalho, foi desenvolvido um programa em C utilizando processos, semáforos e memória compartilhada para a coordenação de múltiplas execuções concorrentes. A implementação do sistema envolveu a criação de uma barreira de sincronização e um mecanismo de controle de acesso exclusivo a um recurso compartilhado, com o uso de uma fila FIFO (First-In, First-Out).

A barreira de sincronização garantiu que todos os processos se sincronizassem antes de prosseguir para a execução do código, assegurando que o programa operasse de forma coordenada. A utilização de semáforos permitiu o controle adequado da execução, evitando condições de corrida e garantindo a correta ordem de execução dos processos.

O mecanismo de controle de acesso exclusivo ao recurso foi implementado por meio de uma fila FIFO, onde os processos aguardam na fila para obter acesso ao recurso compartilhado. Esse controle foi crucial para garantir que os processos acessassem o recurso de forma ordenada e justa, independentemente de sua prioridade ou da política de agendamento do sistema operacional.

Adicionalmente, a implementação de memória compartilhada permitiu o armazenamento das estruturas de dados, como a barreira e a fila FIFO, tornando o programa mais eficiente ao evitar a duplicação de informações entre os processos. O uso de semáforos também foi fundamental para garantir a segurança das operações, protegendo as variáveis compartilhadas contra modificações simultâneas.

Em resumo, o programa desenvolvido demonstrou a eficácia da sincronização entre processos por meio de barreiras e o controle de acesso concorrente a recursos compartilhados, utilizando semáforos e memória compartilhada. O sistema garante uma execução ordenada e eficiente, com a possibilidade de reutilização das estruturas de sincronização, o que possibilita o uso do programa em diferentes cenários e com diferentes números de processos.