Relatório do Segundo Trabalho de Implementação

Problema dos Leitores e Escritores sem inanição

Gabriel de Souza Ferreira
DRE: 115171168

Mayara Martins Poim Fernandes
DRE: 116023936

Introdução

Este trabalho tem como objetivo implementar uma solução em C para o problema dos leitores e escritores de forma que não haja inanição em nenhuma das partes. Para testar se a solução está correta, usamos um script em Python gerado pelo próprio programa C que cria um log que valida cada execução da aplicação. Os componentes básicos do trabalho são: o código em C, o código em Python que define as funções do log de validação e um outro script em Python que testa essas funções de validação.

Organização

A organização dos arquivos é simples: no diretório principal, encontram-se o código em C, um makefile, uma pasta chamada “log”, que contém os arquivos em Python que ajudam a gerar o log de validação, e uma outra chamada “output-leitores”, que armazena os arquivos de texto gerados pelos leitores. O Makefile pode ser usado para executar o programa dos leitores e escritores e seu log de validação de uma só vez.

Algoritmo

Nosso algoritmo obedece a lógica básica do problema dos leitores e escritores (leitores podem ler ao mesmo que outros leitores mas não ao mesmo tempo que escritores escrevem, e apenas um escritor deve escrever de cada vez) ao mesmo tempo que impede que haja inanição em alguma das partes. A lógica básica é implementada com o uso de duas variáveis de estado (uma que conta quantos leitores estão lendo e outra que diz se algum escritor está escrevendo), um mutex e duas variáveis de condição (uma para quando damos um 'cond_wait' nos leitores e outra para quando damos nos escritores). Os leitores operam da seguinte forma:

- Antes de começar a ler, checa se há algum escritor escrevendo.
    - Se um escritor estiver escrevendo, entra no cond_wait.
    - Se não, incrementa a variável que conta os leitores e começa a ler.

- Quando termina de ler, decrementa a variável que conta os leitores e chama a função cond_signal usando a variável de condição dos escritores (libera um escritor).

Os escritores, por sua vez, seguem o seguinte padrão:

- Antes de começar a escrever, checa se há algum leitor lendo ou escritor escrevendo.
    - Se pelo menos um leitor ainda estiver lendo ou um escritor estiver escrevendo, entra no cond_wait.
    - Se não, sinaliza através da variável de estado que está começando a escrever e então começa a escrever.

- Quando termina de escrever, sinaliza através da variável de estado que está terminando de escrever e depois chama as funções cond_signal usando a variável de condição dos escritores e cond_broadcast usando a variável de condição dos leitores.

Complementando essa lógica básica, ainda temos a que impede a inanição. Implementamos essa lógica usando mais três variáveis de estado: uma que diz se existe algum leitor esperando para ler (ou seja, um leitor que está preso no cond_wait), uma que diz se existe algum escritor esperando para escrever e uma que indica se é a vez dos leitores lerem ou se é a vez dos escritores escreverem. Antes dos leitores (escritores) começarem a ler (escrever), eles ainda checam uma última condição para decidir se entram no cond_wait ou se prosseguem:

- Os leitores checam se existe algum escritor esperando para escrever. Caso haja escritores esperando e seja a vez dos escritores, os leitores ficam na espera.

- Os escritores checam se existe algum leitor esperando para ler. Caso haja leitores esperando e seja a vez dos leitores, os escritores ficam na espera.

Além disso, quando um leitor (escritor) termina de ler, ele muda a variável de estado para indicar que é a vez dos escritores (leitores).

Todas as variáveis de estado supracitadas são acessadas de forma atômica com o uso do mutex, evitando condições de corrida.

Além de efetuar o algoritmo descrito, nosso código também atende aos outros requisitos do trabalho. Cada escritor atribui seu id a uma mesma variável global no momento da escrita e cada leitor imprime o valor dessa variável em um arquivo texto a cada vez que faz uma leitura. Cada thread leitora e cada escritora lê/escreve um certo número de vezes (parte do input do programa) antes de terminar seu trabalho e ser recolhida pela thread principal.

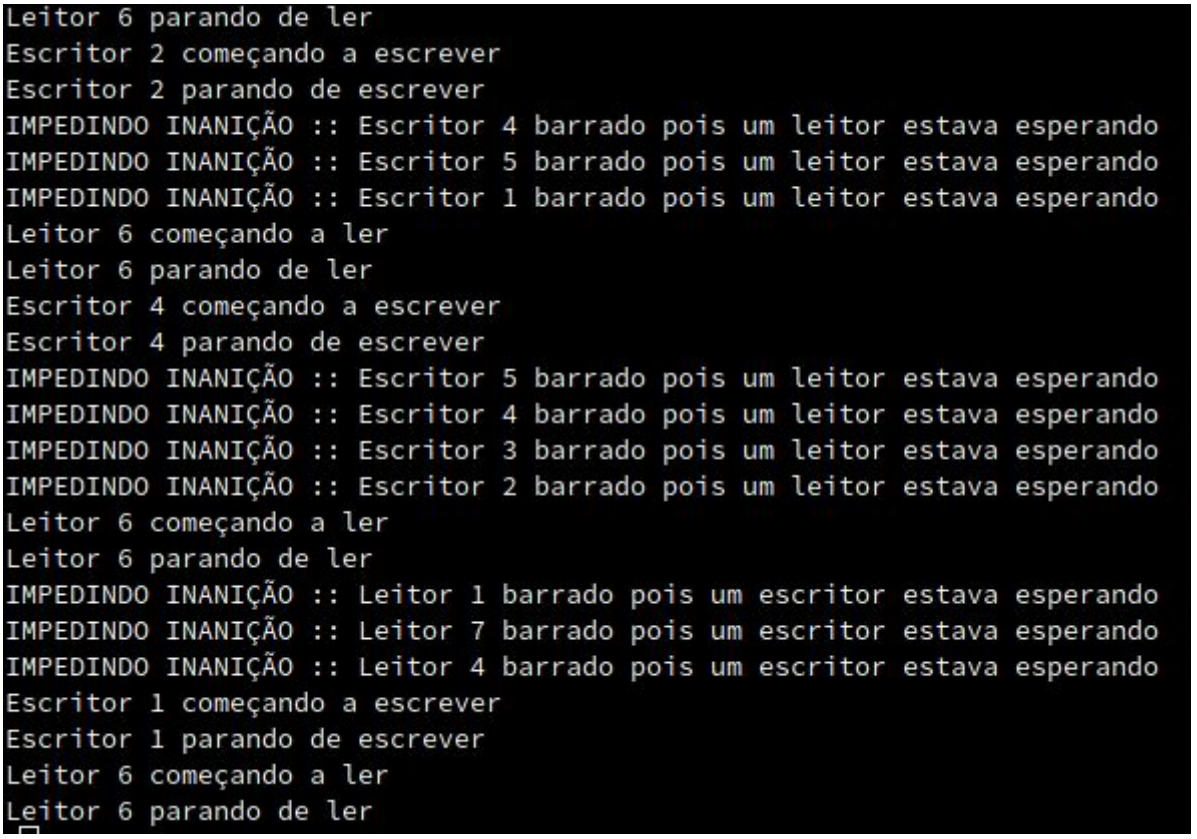
Log

Sempre que um leitor começa ou termina de ler, é bloqueado ou sai do bloqueio, ele imprime seu estado em um arquivo de texto, que é o log de execução (os escritores fazem a mesma coisa). Esse log pode ser lido por um interpretador de Python que irá retornar na saída padrão a ordem em que os eventos aconteceram (quando um escritor foi bloqueado, quando um leitor começou a ler, etc.) e irá indicar se algum dos requisitos do problema foi desrespeitado (se um leitor começou a ler enquanto era a vez de um escritor que estava esperando escrever, por exemplo). Caso o programa concorrente tenha executado sem erros aparentes, o script sinaliza o sucesso no final de suas mensagens. O script chama atenção para quando um escritor é bloqueado pois havia leitores esperando e vice-versa, que são os momentos onde as medidas anti-inanição estão atuando.

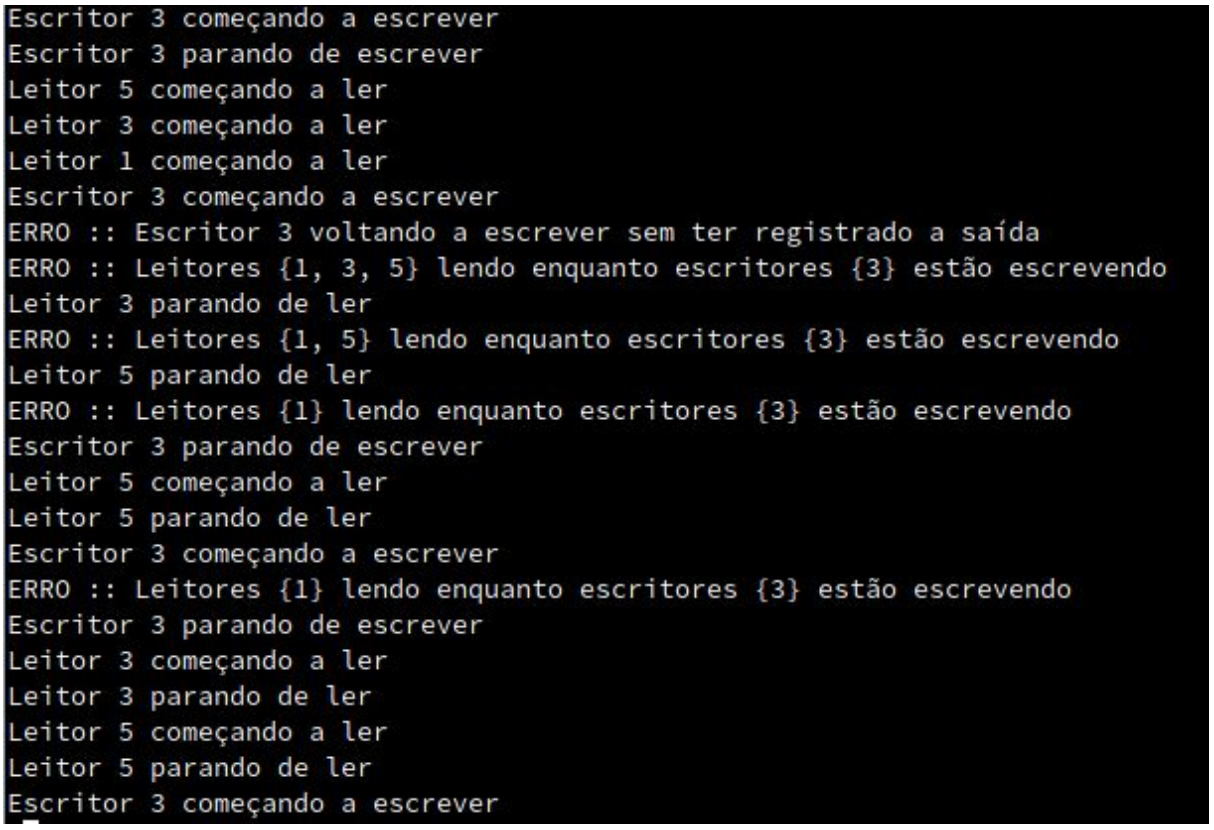
O arquivo de log é formado por chamadas de funções que estão definidas no arquivo “verificador.py”. São funções bastante simples, apenas atualizam o estado dos leitores e escritores e identificam quando entramos em um estado

que não é permitido. Também criamos alguns testes para essas funções com o intuito de checar se elas estavam de fato pegando os erros.

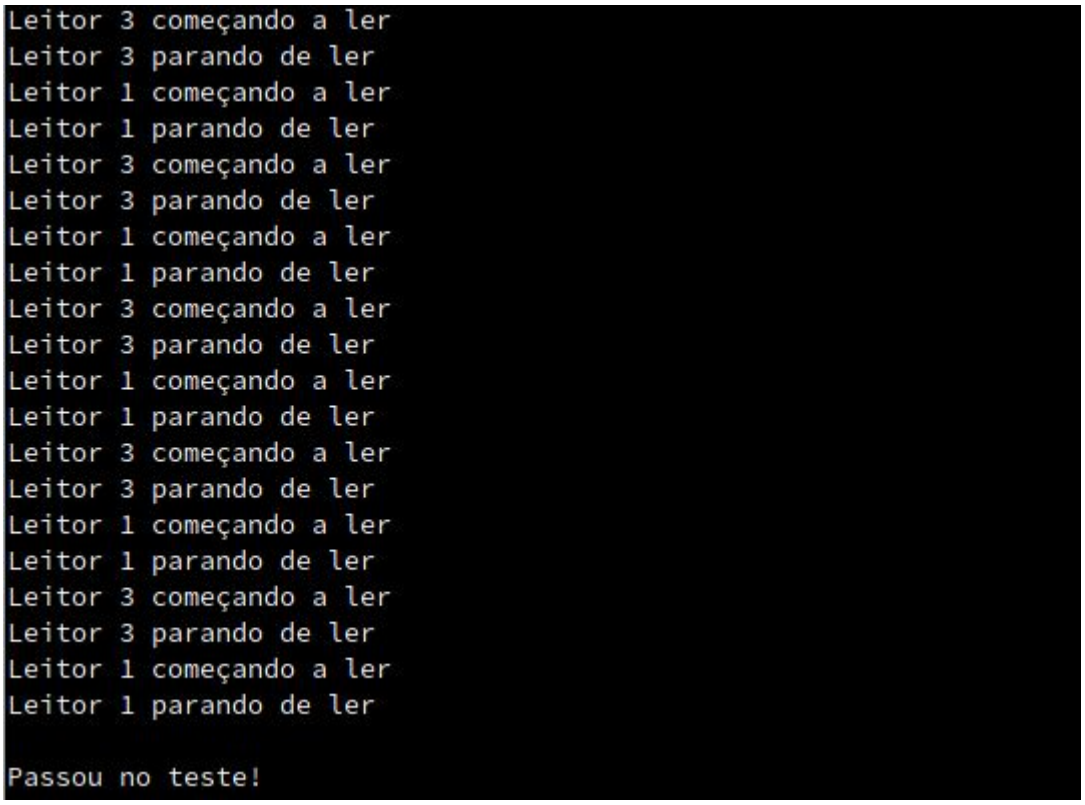
Exemplo de log indicando que o programa evitou inanição:



Exemplo de log com erros (esses erros são propositais, nosso programa finalizado não faz esse tipo de coisa):



Exemplo de log bem sucedido:



Testes

Fizemos vários testes. Mais leitores que escritores, mais escritores do que leitores, tantos leitores quanto escritores, várias vezes, com quantidades variadas. O caso mais extremo testado foi com 30 leitores e 29 escritores, onde cada thread fazia 2000 leituras/escritas (não tentamos nada maior do que isso por conta de limitações do hardware). Não houve erros em nenhum dos testes.

Alguns dos testes realizados:

- 5 leitores e 3 escritores com 20 leituras e 20 escritas cada: 0.0227 seg
- 5 L e 3 E com 2000 leituras e 2000 escritas cada: 1.242 segundos
- 10 L e 10 E com 2000 leituras e 2000 escritas cada: 3.5764 segundos
- 10 L e 20 E com 2000 leituras e 2000 escritas cada: 5.0342 segundos
- 20 L e 10 E com 2000 leituras e 2000 escritas cada: 5.3622 segundos
- 10 L e 10 E com 200 leituras e 2000 escritas cada: 1.7714 segundos
- 10 L e 10 E com 2000 leituras e 200 escritas cada: 1.7698 segundos
- 1000 L 1000 E com 10 leituras e 10 escritas cada: 1.7980 segundos
- 100 L 100 E com 100 leituras e 100 escritas cada: 2.4737 segundos

Os arquivos estão todos no github:

https://github.com/gabrielsferre/trab-leitores-escritores

Além dos arquivos já citados, o repositório possui uma pasta chamada ‘include’ que contém headers com algumas funções auxiliares (alguns wrappers e uma função usada no cálculo do tempo de execução, nada que modifique o funcionamento do programa) . Possui também o README, com instruções de como executar a aplicação.