UFRJ

# Trabalho 2 - Leitores e Escritores

Computação Concorrente

Alunos:
Victor Cruz - 114079525
Marcos Eduardo - 114097955

# Leitores e Escritores sem Inanição

Neste trabalho projetamos e implementamos uma solução para o problema dos leitores e escritores utilizando a linguagem C. Codificamos uma solução simples para o problema que garante a inanição entre as threads, ou seja, não há "starvation".

Além disso, codificamos também um programa auxiliar em python que verifica a corretude do nosso programa principal. Para cada ação, o programa em C escreve um comando (ex: "criaLeitor()") em um arquivo .txt, com nome escolhido pelo usuário. Ou seja, o programa auxiliar nada mais é que um arquivo python com várias de funções que são executadas na ordem em que aparecem no arquivo .txt. Ao final, este programa imprime na tela se o programa principal foi executado corretamente ou não.

Valer ressaltar que todos os códigos foram escritos em um macOS, o que trouxe uma complicação. A biblioteca de semáforos utilizada em aula "semaphore.h" não funciona neste OS. Ao tentar utilizá-la, recebemos avisos em tempo de compilação de que as rotinas estão depreciadas. Ao executar, notamos que os semáforos simplesmente não funcionam. Provamos isso ao setar todos os semáforos com valor inicial 0 e nenhuma thread foi interrompida ao chamar "wait". Para resolver esse problema, utilizamos outra biblioteca chamada "dispatch.h". Ela possui uma implementação de rotinas com argumentos um pouco diferentes, mas durante todo o desenvolvimento do trabalho, ela se mostrou com um comportamento similar ao da outra biblioteca e suficiente para atingirmos o objetivo da implementação.

# Características da Implementação

Nosso programa conta com algumas características, como, por exemplo, leitores e escritores têm mesma prioridade, também conhecido como o terceiro problema dos leitores e escritores (o primeiro e o segundo causam starvation). Fizemos desta forma para poder garantir a inexistência de starvation, assim leitores e escritores terão acesso à variável compartilhada em sua ordem de chegada. Isso ocorre pois a implementação dos semáforos utiliza filas com o algoritmo FIFO.

Por motivos de simplicidade, em nosso programa também não é necessário que um escritor seja o primeiro a executar. Além destas, garantimos também as exigidas no roteiro, como mais de um leitor poder ler ao mesmo tempo, apenas um escritor pode escrever de cada vez e não é permitido ler e escrever ao mesmo tempo.

Para conseguirmos controlar a execução do programa de forma a satisfazer os requisitos, utilizamos algumas das estruturas estudadas  em sala de aula, como semáforos para garantir a inanição e "locks" para garantir a exclusão mútua em áreas críticas.

# Implementação

## - Thread Escritora:

```
void *Escritor (void *arg) {
    int idThread = *(int *) arg;

    int escritas = NEscritas;
    while(escritas > 0) {
        dispatch_semaphore_wait(inan, DISPATCH_TIME_FOREVER);
        dispatch_semaphore_wait(acesso, DISPATCH_TIME_FOREVER);
        dispatch_semaphore_signal(inan);
```

Esta é a primeira parte da thread escritora. Ela recebe seu id por argumento e pega de uma variável global (preenchida pela linha de comando) o número de escrituras que ela deve realizar. Antes de escrever, ela realiza um wait e um signal no semáforo de inanição, com um wait no semáforo de acesso entre eles. Desta forma podemos fazer com que as threads obedeçam a ordem de execução.

```
        dispatch_semaphore_signal(acesso);
        escritas--;
    }

    free(arg);
    pthread_exit(NULL);
}
```

Após escrever seu id na variável compartilhada, a thread libera o acesso, acaba seu loop e termina.

- Thread Leitora:

```
void *Leitor (void *arg) {
    int idThread = *(int *) arg;

    FILE *arquivo;
    int tamanho = snprintf(NULL, 0,"%d.txt", idThread);
    char nome[tamanho + 1];
    sprintf(nome, "%d.txt", idThread);
    arquivo = fopen(nome, "w");

    int leituras = NLeituras;
    while(leituras > 0) {
        dispatch_semaphore_wait(inan, DISPATCH_TIME_FOREVER);
        pthread_mutex_lock(&l_mutex); l++;
        if(l==1) {
            dispatch_semaphore_wait(acesso, DISPATCH_TIME_FOREVER);
            fprintf(arquivoDeLog, "pedeAcessoParaLer(%d)\n", idThread);
        }
        dispatch_semaphore_signal(inan);
        pthread_mutex_unlock(&l_mutex);
```

A thread leitora também recebe seu id por argumento. Ela abre o arquivo onde irá escrever o conteúdo da variável compartilhada. Assim como a escritora, ela pega de uma variável global o número de loops que irá realizar. Dentro do loop, assim como a escritora, ela realiza um wait e um signal no semáforo de inanição e entre essas duas chamadas ela pede acesso. Porém, como apenas o primeiro leitor precisa pedir esse acesso, temos uma variável de controle com um if para verificar se esta thread é a primeira a executar. Para conseguirmos utilizar essa variável, precisamos usar um lock para garantir a exclusão mútua.

```
        pthread_mutex_lock(&l_mutex); l--;
        if(l==0) {
            fprintf(arquivoDeLog, "liberaAcessoDaLeitora(%d)\n", idThread)
            dispatch_semaphore_signal(acesso);
        }
        pthread_mutex_unlock(&l_mutex);

        leituras--;
    }

    fclose(arquivo);
    free(arg);
    pthread_exit(NULL);
}
```

Após ler o conteúdo da variável de compartilhada e escrever este conteúdo em seu arquivo, a thread precisa realizar o mesmo processo de acessar a variável de controle por exclusão mútua e liberar o acesso caso seja a última a executar. Depois, a thread acaba seu loop e termina.

## Analisando os Outputs

```
Victors-MacBook-Pro:trab2 victorcruz$ ./trab2
Use: ./trab2 <numero de leitores> <numero de escritores> <numero de leituras> <numero
de escritas> <nome do arquivo de log>
```

O programa possui um pequeno guia do que ele espera, caso o usuário não saiba como executá-lo.

```
Victors-MacBook-Pro:trab2 victorcruz$ ./trab2 10 10 1 1 log.txt
Escritor 10 escreveu
Leitor 1 leu
Leitor 3 leu
Leitor 8 leu
Escritor 11 escreveu
Leitor 6 leu
Escritor 12 escreveu
Leitor 2 leu
Escritor 13 escreveu
Leitor 4 leu
Leitor 0 leu
Leitor 5 leu
Leitor 7 leu
Escritor 14 escreveu
Leitor 9 leu
Escritor 15 escreveu
Escritor 16 escreveu
Escritor 17 escreveu
Escritor 18 escreveu
Escritor 19 escreveu

FIM
```

```
Victors-MacBook-Pro:trab2 victorcruz$ ./trab2 10 10 1 1 log.txt
Leitor 0 leu
Leitor 4 leu
Leitor 3 leu
Leitor 1 leu
Leitor 5 leu
Escritor 11 escreveu
Escritor 10 escreveu
Leitor 6 leu
Escritor 12 escreveu
Escritor 13 escreveu
Escritor 14 escreveu
Leitor 2 leu
Escritor 15 escreveu
Escritor 16 escreveu
Leitor 8 leu
Leitor 7 leu
Leitor 9 leu
Escritor 18 escreveu
Escritor 19 escreveu
Escritor 17 escreveu

FIM
```

Podemos ressaltar, primeiramente, que os argumentos passados pela linha de comando são respectivamente o número de threads leitoras,o número de threads escritoras, o número de leituras realizadas por thread, o número de escritas realizadas por thread e o nome do arquivo onde serão escritos os logs do programa principal.

Após a execução, vemos que o programa imprime cada vez que uma thread realiza sua ação principal, referenciada pelo seu id. Podemos notar que não há um padrão para a ordem em que as thread são executadas. Porém podemos também afirmar que na maioria das vezes provavelmente alguma thread escritora começará antes de uma leitora já que as leitoras demandam tempo para abrir seus arquivos de escrita.

```
Escritor 4 escreveu
Escritor 6 escreveu
Leitor 0 leu
Escritor 3 escreveu
Escritor 5 escreveu
Leitor 2 leu
Escritor 4 escreveu
Leitor 1 leu
Escritor 6 escreveu
Leitor 0 leu
Escritor 3 escreveu
Escritor 5 escreveu
Leitor 2 leu
Escritor 4 escreveu
Leitor 1 leu
Escritor 6 escreveu
Leitor 0 leu
Escritor 3 escreveu
Escritor 5 escreveu
Leitor 2 leu
Escritor 4 escreveu
Leitor 1 leu
Escritor 6 escreveu
Leitor 0 leu
Escritor 3 escreveu
```

Quando colocamos mais leituras e escritas por threads e menos threads podemos observar algo curioso. Neste caso parece que há uma ordem de execução que se repete. Isso se deve ao fato de que os semáforos implementam uma fila de espera com algoritmo "first in first out". Então quando alguma thread chama a rotina signal, a primeira thread a chamar wait é liberada, criando assim essa repetição. Garantimos dessa forma que nenhuma thread cairá na situação de starvation. Caso contrário, não poderíamos garantir isso, já que as threads executadas seriam liberadas do wait em ordem aleatória.

## Programa Auxiliar

```
numeroDeLeitores = 0
numeroDeEscritores = 0
numeroEsperadoDeLeituras = 0
numeroEsperadoDeEscritas = 0
numeroRealizadoDeLeituras = 0
numeroRealizadoDeEscritas = 0
qualEscritorEstaEscrevendo = -1
ultimaThreadAEscrever = -1
quantidadeDeLeitoresLendo = 0
escritaPermitida = True
leituraPermitida = True

numeroDeFalhas = 0

def criaLeitor():
    global numeroDeLeitores
    numeroDeLeitores += 1

def criaEscritor():
    global numeroDeEscritores
    numeroDeEscritores += 1

def registraNumeroEsperadoDeLeituras(numero):
    global numeroEsperadoDeLeituras, numeroDeLeitores
    numeroEsperadoDeLeituras = numero * numeroDeLeitores

def registraNumeroEsperadoDeEscritas(numero):
    global numeroEsperadoDeEscritas, numeroDeLeitores
    numeroEsperadoDeEscritas = numero * numeroDeLeitores

def pedeAcessoParaEscrever(id):
    global numeroDeFalhas, leituraPermitida
    if not leituraPermitida or quantidadeDeLeitoresLendo > 0:
        numeroDeFalhas += 1
        print("Falha em pedir acesso para escrita pela thread {}".format(id))
    else:
        leituraPermitida = False
```

O programa auxiliar tenta reproduzir o contexto do programa principal com as variáveis mostradas acima. Todas as rotinas descritas neste programa alteram as variáveis de uma forma que outras rotinas avisem um possível erro caso sejam invocadas numa ordem inesperada.
Por exemplo, se a rotina "le(1)" é chamada, significa que a thread com id 1 está lendo da variável compartilhada. Se logo após isso, a rotina "escreve(2)" for chamada, algo está errado, pois o acesso não foi liberada pela thread 1.

```
Victors-MacBook-Pro:trab2 victorcruz$ python aux.py
Falha em parar a escrita, ela nao estava escrevendo. Thread 11
Falha em liberar acesso da escritora, ela nao tinha acesso. Thread 11
Falha em pedir acesso para escrita pela thread 10
Falha em parar a escrita, ela nao estava escrevendo. Thread 10
Falha em liberar acesso da escritora, ela nao tinha acesso. Thread 10
Falha em pedir acesso para leitura pela thread 6
Falha em ler, sem acesso. Thread 10
```

Quando há algum problema devido a ordem de execução das rotinas, o programa irá imprimir os avisos de erro devidamente. Após uma rotina falhar, ela não realizará a alteração das variáveis de contexto devidamente, então a execução das rotinas seguintes não deverão funcionar, como em um efeito dominó.

Ao fim, caso tudo tenha corrido bem, o programa auxiliar imprime uma mensagem de sucesso.

```
Victors-MacBook-Pro:trab2 victorcruz$ python aux.py
Parabens, o programa executou como o esperado!
```

## Bibliografia


- https://rfc1149.net/blog/2011/01/07/the-third-readers-writers-problem/
- https://en.wikipedia.org/wiki/Readers–writers_problem#Algorithm