

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO



UFRJ

Daniel Lopes de Sousa

DRE: 114143976

Lucas Clemente Miranda Braga

DRE: 115159700

Leitores e escritores sem inanição:
Segundo Trabalho
Disciplina de Computação Concorrente

Professora: Silvana Rossetto

Sumário

1. Introdução	3
1.1 Objetivos	3
1.2 Tecnologias utilizadas	3
2. Organização do projeto	4
2.1 Solução Concorrente	4
2.1.1 Estruturas e funções geradas	6
2.2 Programa auxiliar	10
2.2.1 Estruturas e funções geradas	12
3. Testes realizados	16
4. Conclusão	18
5.1 Problemas e dificuldades encontrados	18
5.2 Alcance dos objetivos	18

1. Introdução

Este relatório tem como objetivo documentar o desenvolvimento do segundo trabalho feito para disciplina de Computação Concorrente no período de 2019-2 do curso de Ciência da Computação da Universidade Federal do Rio de Janeiro.

1.1 Objetivos

O principal objetivo do presente trabalho é o de aprendizado na prática de conceitos teóricos aprendido em sala. Aprendizado este que proporciona a oportunidade de crescimento como alunos e desenvolvedores.

Além disso, o projeto tem como objetivo implementar uma solução para o problema de leitores e escritores, um problema clássico na computação concorrente. As condições lógicas gerais do problema são:

- vários leitores podem ler simultaneamente;
- somente um escritor pode escrever por vez;
- leitura e escrita não pode acontecer simultaneamente.

Como recurso compartilhado entre as threads, temos uma variável de valor inteiro. Cada thread recebe como identificador também um número inteiro. As threads leitoras escrevem seus identificadores em um arquivo *.txt* único, que recebe como nome o próprio identificador de cada uma. Ou seja, teremos arquivos *1.txt*, *2.txt*, *3.txt* e assim por diante. Já as threads escritoras escrevem o seu identificador na variável compartilhada.

Adicionalmente às condições do problema já mencionadas, temos como condição a garantia de ausência de inanição (ou *starvation*) de threads na solução do problema. Por isso, podemos caracterizá-la como objetivo final do trabalho.

1.2 Tecnologias utilizadas

Para a implementação do projeto foram utilizadas as linguagens de programação C e Python, juntamente com as bibliotecas relacionadas ao uso e controle de threads (“*pthread*”) e semáforos

(“*semaphore*”), ambas bibliotecas padrão da linguagem C. Sendo o uso de C para a implementação da solução concorrente e de Python para o programa auxiliar para verificar a corretude da solução concorrente.

2. Organização do projeto

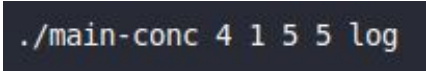
Nesta seção está especificada a solução concorrente para o problema juntamente com o programa auxiliar para verificar a corretude desta solução (estruturas e funções utilizadas e a lógica dos programas).

A solução concorrente implementada gera como saída um arquivo de log, contendo todos os passos realizados pelas threads durante a execução do programa. O programa auxiliar já citado analisa este arquivo de log e executa os passos descritos nele de forma sequencial, a fim de verificar se, em algum, passo o programa cometeu algum erro de acordo com as definições do problema. Além disso, o programa auxiliar verifica a existência de inanição de alguma das threads.

2.1 Solução Concorrente

Para a solução concorrente, a função principal recebe como parâmetros da linha de comando a quantidade de threads leitoras e escritoras, a quantidade de leituras e escritas e o nome do arquivo de log. A leitura é feita como *<nome do programa> <número de threads leitoras> <número de threads escritoras> <número de leituras> <número de escritas> <nome do arquivo de log>*, onde o número de threads leitoras é armazenado em *nThreadsLeitoras* e o número de threads escritoras é armazenado em *nThreadsEscritoras*. Já os valores de leituras, escritas e o nome do arquivo de log são armazenados em *nDeLeituras*, *nDeEscritas* e *nomeDoArquivo* respectivamente.

Abaixo segue um exemplo de entrada do usuário para o programa e o trecho de código em que é feita a leitura dessa entrada e armazenados os valores nas variáveis correspondentes.



```
./main-conc 4 1 5 5 log
```

Figura 1. Exemplo de entrada do programa sequencial.

Entrando no escopo do programa, temos a definição das variáveis globais que incluem as já citadas recebidas como entrada pela linha de comando, além de algumas outras. Entre elas, temos os

contadores do número de leituras e escritas realizadas pelo programa, o número de leitores e escritores em processo de leitura e escrita, além do recurso compartilhado pelas threads, as variáveis de exclusão mútua por lock e os semáforos a serem utilizados nas funções executadas pelas threads e, finalmente, as variáveis dos arquivos de saída.

```
int nDeEscritas, nDeLeituras;
int leiturasFeitas = 0, escritasFeitas = 0;
int nThreadsEscritoras, nThreadsLeitoras;
int leitores = 0, escritores = 0;
int leitor = -1, escritor = -1;
int recurso = -1;
int threads = 0;
pthread_mutex_t readerMutex, barrierMutex;
pthread_cond_t cond;
sem_t em, fila;

FILE ** files;
FILE * logger;
```

Figura 2. Variáveis globais do programa.

Entrando na main temos as definições de variáveis auxiliares para as threads e arquivos, além da leitura dos argumentos pela linha de comando e atribuição desses valores a suas respectivas variáveis e a inicialização dos semáforos e locks.

```
int i;
int * idThread;
char * nomeDoArquivo;
pthread_t * tid;
FILE * tempFile;

if(argc < 6) {
    printf("\nDigite: %s numeroThreadsLeitoras numeroThreadsEscritoras
    numeroLeituras numeroEscritas nomeArquivoLog\n\n", argv[0]); exit(-1);
}

pthread_mutex_init(&readerMutex, NULL);
pthread_mutex_init(&barrierMutex, NULL);
pthread_cond_init(&cond, NULL);
sem_init(&em, 0, 1);
sem_init(&fila, 0, 1);

nThreadsLeitoras = atoi(argv[1]);
nThreadsEscritoras = atoi(argv[2]);
nDeLeituras = atoi(argv[3]);
nDeEscritas = atoi(argv[4]);
nomeDoArquivo = argv[5];
```

Figura 3. Definição de variáveis auxiliares e preenchimento das variáveis globais e de arquivo.

Em seguida, criamos os arquivos que serão utilizados para escrever as leituras das threads leitoras e o log do programa.

Utilizamos no programa concorrente em C um arquivo auxiliar e temporário, com o único objetivo de armazenar o nome do arquivo de log para que este seja acessado pelo programa auxiliar. Isto porque no arquivo de log temos apenas chamadas de função definidas no programa auxiliar, mas para executá-las o programa precisa abrir o arquivo de log inicialmente, mas o nome deste arquivo é mutável porque é recebido pela linha de comando do programa concorrente. Assim, definimos um arquivo temporário de nome fixo, onde o programa auxiliar lê o nome do arquivo de log definido pelo usuário para iniciar sua execução com a primeira chamada de função *iniciaVerificacao()*.

As funções do arquivo de verificação serão definidas mais para frente no escopo deste relatório.

```
files = malloc (sizeof(FILE *) * nThreadsLeitoras);

strcat(nomeDoArquivo, ".txt");
logger = fopen(nomeDoArquivo, "w");
tempFile = fopen("temp.txt", "w");

fprintf(tempFile, "%s", nomeDoArquivo);

//printa no arquivo de log os parametros iniciais do programa passados na linha de comando
fprintf(logger, "iniciaVerificacao(%d, %d, %d, %d)\n",
nThreadsLeitoras, nThreadsEscritoras, nDeLeituras, nDeEscritas);
```

Figura 4. Criação dos arquivos e primeira escrita no arquivo de log.

Depois disso, as threads são criadas e lançadas e o programa espera que elas terminem para encerrar sua execução. Este programa concorrente não produz nenhuma saída no terminal, apenas um aviso de que a execução foi concluída, sendo suas saídas principais os arquivos preenchidos.

2.1.1 Estruturas e funções geradas

Na solução concorrente do projeto, temos definidas as funções executadas pelas threads leitoras e escritoras, além das funções de leitura e escrita e uma função de barreira.

Primeiro definimos a função *barreira()*, cujo objetivo é impedir que as threads iniciem sua

execução até que todas tenham sido lançadas e iniciadas. Para isso, foi feita uma sincronização por condição.

```
pthread_mutex_lock(&barrierMutex);
threads++;
if (threads < (nThreadsEscritoras + nThreadsLeitoras)) {
    pthread_cond_wait(&cond, &barrierMutex);
}
else {
    pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&barrierMutex);
```

Figura 5. Escopo da função *barreira()*.

Em seguida, as definições das funções *Leitura()* e *Escrita()*, ambas recebendo o id da thread que chamou a leitura/escrita como parâmetro.

Na função de escrita temos apenas uma linha que atribui o valor do id da thread à variável compartilhada. Já na função de leitura, temos uma escrita no arquivo corresponde à thread que especifica o valor lido por ela.

```
recurso = idThreadEscritora;
```

Figura 6. Escopo da função *Escrita()*.

```
fprintf(files[idThreadLeitora - nThreadsEscritoras],
"Thread %d leu o valor %d\n", idThreadLeitora, recurso);
return 0;
```

Figura 7. Escopo da função *Leitura()*.

```
Thread 4 leu o valor -1
Thread 4 leu o valor 0
Thread 4 leu o valor 0
Thread 4 leu o valor 0
```

Figura 8. Exemplo de arquivo de saída de leitor.

Temos então a definição das funções que serão executadas pelas threads *threadLeitora()* e *threadEscritora()*. Ambas recebendo como parâmetro o identificador da thread.

Começamos pela função das threads escritoras. Aqui devemos explicar a função dos semáforos utilizados (*em* e *fila*). O semáforo *em* foi utilizado para fazer a exclusão mútua entre as threads leitoras e escritoras, garantindo que não haja execução dos dois tipos de threads simultaneamente. O segundo semáforo, *fila*, foi utilizado para evitar inanição de um tipo de threads, porque garante prioridade para os dois tipos.

Assim, quando um escritor entra para execução, ele pega os semáforos de exclusão mútua e fila, liberando o de fila assim que consegue pegar o de exclusão mútua. E ao terminar a execução, libera o semáforo de exclusão mútua.

```
int * tid = (int * ) id;
barreira();

while(escritasFeitas < nDeEscritas){;

    sem_wait(&fila);
    sem_wait(&em);
    escritor = *tid;
    if(leitores > 0) {
        fprintf(logger, "leituraBloqueada(%d)\n", leitor);
    }
    fprintf(logger, "entraEscrita(%d)\n", *tid);
    sem_post(&fila);

    Escrita(*tid);

    escritasFeitas++;
    fprintf(logger, "saiEscrita(%d)\n", *tid);

    sem_post(&em);
}

free(tid);
pthread_exit(NULL);
```

Figura 9. Escopo da função *threadEscritora()*.

Na função de leitores, foi utilizado também um lock, mas apenas para incrementar a variável *leitores* sem condição de corrida, visto que podemos ter mais de um leitor executando ao mesmo tempo.

```
int * tid = (int * ) id;
barreira();

while(leiturasFeitas < nDeLeituras){

    sem_wait(&fila);
    pthread_mutex_lock(&readerMutex);
    if(leitores == 0){ sem_wait(&em); }
    leitor = *tid;
    if(escritores > 0) {
        fprintf(logger, "escritaBloqueada(%d)\n", escritor);
    }
    fprintf(logger, "entraLeitura(%d)\n", *tid);
    leitores++;
    sem_post(&fila);
    pthread_mutex_unlock(&readerMutex);

    Leitura(*tid);

    pthread_mutex_lock(&readerMutex);
    leitores--;
    leiturasFeitas++;
    fprintf(logger, "saiLeitura(%d, %d)\n", *tid, recurso);
    if(leitores == 0) { sem_post(&em); }
    pthread_mutex_unlock(&readerMutex);
}

free(tid);
pthread_exit(NULL);
```

Figura 10. Escopo da função *threadLeitora()*.

Além disso, temos nas duas funções escritas no arquivo de log, referentes aos passos sendo executados no programa. Ao fim da execução, temos o arquivo de log preenchido com diversas chamadas de função que representam as leituras e escritas que serão executadas pelo programa auxiliar.

```

iniciaVerificacao(4, 1, 5, 5)
entraLeitura(4)
entraLeitura(1)
saiLeitura(4, -1)
saiLeitura(1, -1)
entraEscrita(0)
saiEscrita(0)
entraEscrita(0)
saiEscrita(0)
entraEscrita(0)
saiEscrita(0)
entraLeitura(4)
saiLeitura(4, 0)
entraLeitura(4)
saiLeitura(4, 0)
entraLeitura(4)
saiLeitura(4, 0)
entraEscrita(0)
saiEscrita(0)
entraEscrita(0)
saiEscrita(0)
entraLeitura(1)
saiLeitura(1, 0)
entraLeitura(2)
saiLeitura(2, 0)
entraLeitura(3)
saiLeitura(3, 0)

```

Figura 11. Exemplo de arquivo de log.

2.2 Programa auxiliar

No programa auxiliar temos as definições das funções que farão a verificação de cada etapa da execução do programa concorrente. Para fazer isso, o programa define variáveis similares às definidas no programa concorrente, a fim de simular sua execução de forma sequencial.

Assim, definimos variáveis para o número total de threads e o número de leitores e escritores, além do recurso compartilhado e uma lista que guarda os ids das threads que foram executadas. Definimos também uma variável para o arquivo que será aberto para leitura das chamadas de função.

```
# variaveis globais
nThreads = 0
totalLeitores = 0
totalEscritores = 0
leitores = 0
escritores = 0
recurso = -1

arquivoAux = open("temp.txt", "r")
arquivo = open(arquivoAux.readline(), "r")

threadsExecutadas = []
```

Figura 12. Variáveis globais do programa auxiliar.

Depois dessa definição temos a definição de todas as funções de verificação (que serão explicitadas no próximo item), seguidas da leitura do arquivo para executar as suas linhas.

Uma vez executadas as linhas e verificado se existem erros na leitura ou escrita, é feita por último a verificação de inanição de threads. Sempre que uma nova thread, que não foi executada anteriormente, entra na leitura ou escrita, seu identificador é guardado na lista *threadsExecutadas*, como será mostrado mais adiante. Assim, ao final de toda execução, essa lista é percorrida e, caso um dos identificadores não esteja presente nela, significa que a thread correspondente a ele não realizou nenhuma leitura ou escrita e, portanto, esteve em estado de inanição por toda a execução.

```
# Executa as linhas do arquivo
for linha in arquivo:
    exec(linha)

for i in range(0, nThreads):
    if i not in threadsExecutadas:
        erro(i, 0, 6)

print("\nExecucao finalizada sem erros!\n")
```

Figura 13. Execução das linhas do arquivo de log e última verificação de inanição.

2.2.1 Estruturas e funções geradas

Chegamos então às funções definidas para verificação da execução do programa concorrente. Aqui definiremos cada uma delas e seu escopo.

- *iniciaVerificacao()*:

Primeiro, temos a função que inicia a execução. Ela recebe como parâmetros os números de leitores e escritores, além do número de leituras e escritas. Em seu escopo, estes valores são atribuídos às variáveis globais do programa.

```
def iniciaVerificacao(nLeitores, nEscritores, leituras, escritas):  
    global totalLeitores, totalEscritores  
    global nThreads  
  
    totalLeitores = nLeitores  
    totalEscritores = nEscritores  
    nThreads = nLeitores + nEscritores
```

Figura 14. Escopo da função *iniciaVerificacao()*.

- *erro()*:

Em seguida, a função que é chamada sempre que um erro é encontrado pelas outras funções ou pelo escopo global do programa. Esta função recebe como parâmetros o identificador da thread que a chamou, o tipo desta thread e o código do erro.

Seu escopo consiste apenas de retornar no terminal uma mensagem da thread que executou com erro e do erro que ela encontrou, encerrando a execução do programa em seguida.

```
def erro(t_id, tipoThread, codErro):
    if(tipoThread == 1):
        print("Thread leitora ", t_id, " executou com erro:")
    elif(tipoThread == 2):
        print("Thread escritora ", t_id, " executou com erro:")

    if(codErro == 1):
        print("Mais de um escritor escrevendo ao mesmo tempo")
    elif(codErro == 2):
        print("Leitura e escrita acontecendo simultaneamente")
    elif(codErro == 3):
        print("Leitura bloqueada sem que haja escritores executando")
    elif(codErro == 4):
        print("Escrita bloqueada sem que haja leitores/escritores executando")
    elif(codErro == 5):
        print("Valor lido do recurso diferente do esperado")
    elif(codErro == 6):
        print("Inanicao")

    sys.exit()
```

Figura 15. Escopo da função *erro()*.

- *entraLeitura()* e *saiLeitura()*:

Temos então as funções de simulação de leitura, uma para entrada e outra para saída.

Ao entrar na leitura, recebemos como parâmetro o identificador da thread leitora e o valor global do número de leitores em execução é incrementado. Em seguida, é verificado se existe algum escritor sendo executado e, caso haja, a função de erro é chamada, porque não podem haver leitores e escritores sendo executados ao mesmo tempo.

A função de saída recebe como parâmetros, além do identificador da thread, o valor lido por ela na leitura. Depois, é verificado se o valor lido é o mesmo que o do recurso compartilhado e, se não, significa que houve um erro na leitura. Caso o valor esteja correto, o número de leitores é decrementado e, caso a thread ainda não esteja na lista de threads executadas, ela é adicionada.


```

def entraLeitura(idThread):
    global leitores, escritores
    leitores += 1

    if(escritores > 0):
        erro(idThread, 1, 2)

def saiLeitura (idThread, valorLido):
    global leitores

    if(recurso != valorLido):
        erro(idThread, 1, 5)

    leitores -= 1

    if idThread not in threadsExecutadas:
        threadsExecutadas.append(idThread)

```

Figura 16. Escopo das funções *entraLeitura()* e *saiLeitura()*.

- *entraEscrita()* e *saiEscrita()*:

Temos então as funções de simulação da escrita. Quando uma thread entra na escrita, o valor dos escritores é incrementado e são feitas duas verificações. Uma para o caso de haver outros escritores estarem escrevendo, e outra para o caso de outros leitores estarem lendo simultaneamente à thread atual. Caso qualquer um dos dois seja verdade, temos um erro e função de erro é chamada.

Ao sair da escrita, o recurso é definido como o identificador da thread escritora, o número de escritores é decrementado e a função verifica se a thread atual já está na lista de executadas e a adiciona caso não esteja.

```
def entraEscrita(idThread):
    global leitores, escritores
    escritores += 1

    if(escritores > 1):
        erro(idThread, 2, 1)

    if(leitores > 0):
        erro(idThread, 2, 2)

def saiEscrita(idThread):
    global escritores, recurso

    recurso = idThread
    escritores -= 1

    if idThread not in threadsExecutadas:
        threadsExecutadas.append(idThread)
```

Figura 17. Escopo das funções *entraEscrita()* e *saiEscrita()*.

- *leituraBloqueada()* e *escritaBloqueada()*:

Estas duas funções são chamadas quando as funções de escrita e leitura do programa concorrente são bloqueadas antes de executar. Isso acontece quando um leitor quer ler mas já existe um escritor escrevendo. Ou quando um escritor quer realizar escrita mas já existe um leitor realizando leitura ou outro escritor realizando escrita. Assim, essas funções verificam se essas condições de bloqueio estão sendo atendidas e, se não, chamam a função de erro.

```
def leituraBloqueada(idThread):
    global escritores

    if(escritores == 0):
        erro(idThread, 1, 3)

def escritaBloqueada(idThread):
    global leitores

    if(leitores == 0 and escritores == 0):
        erro(idThread, 2, 4)
```

Figura 18. Escopo das funções *leituraBloqueada()* e *escritaBloqueada()*.

3. Testes realizados

Temos nesta seção alguns exemplos de testes aplicados no programa concorrente, incluindo as informações de entrada e o arquivo de log de saída do algoritmo.

```
./main-conc 2 2 10 10 log
```

Figura 19. Exemplo de entrada do programa concorrente.

```
iniciaVerificacao(2, 2, 10, 10)
entraLeitura(3)
saileitura(3, -1)
entraEscrita(1)
saiEscrita(1)
entraEscrita(1)
saiEscrita(1)
entraLeitura(2)
saileitura(2, 1)
entraEscrita(0)
saiEscrita(0)
entraEscrita(0)
saiEscrita(0)
entraEscrita(1)
saiEscrita(1)
entraEscrita(1)
saiEscrita(1)
entraLeitura(2)
saileitura(2, 1)
entraEscrita(0)
saiEscrita(0)
entraEscrita(0)
saiEscrita(0)
entraEscrita(1)
saiEscrita(1)
entraEscrita(1)
saiEscrita(1)
entraLeitura(2)
saileitura(2, 1)
entraLeitura(3)
saileitura(3, 1)
entraLeitura(3)
saileitura(3, 1)
entraEscrita(0)
saiEscrita(0)
entraLeitura(2)
saileitura(2, 0)
entraLeitura(2)
saileitura(2, 0)
entraLeitura(2)
saileitura(2, 0)
entraLeitura(2)
saileitura(2, 0)
entraLeitura(3)
saileitura(3, 0)
```

Figura 20. Exemplo de log de saída do programa concorrente.


```
Execucao finalizada sem erros!
```

Figura 21. Exemplo de saída do programa auxiliar para o arquivo de log acima.

Segue outro exemplo:

```
./main-conc 10 2 5 5 log
```

Figura 22. Exemplo de entrada do programa concorrente.

```
iniciaVerificacao(2, 2, 10, 10)
entraLeitura(3)
saiLeitura(3, -1)
entraEscrita(1)
saiEscrita(1)
entraEscrita(1)
saiEscrita(1)
entraLeitura(2)
saiLeitura(2, 1)
entraEscrita(0)
saiEscrita(0)
entraEscrita(0)
saiEscrita(0)
entraEscrita(1)
saiEscrita(1)
entraEscrita(1)
saiEscrita(1)
entraLeitura(2)
saiLeitura(2, 1)
entraEscrita(0)
saiEscrita(0)
entraEscrita(0)
saiEscrita(0)
entraEscrita(1)
saiEscrita(1)
entraEscrita(1)
saiEscrita(1)
entraLeitura(2)
```

```
saiLeitura(2, 1)
entraLeitura(3)
saiLeitura(3, 1)
entraLeitura(3)
saiLeitura(3, 1)
entraEscrita(0)
saiEscrita(0)
entraLeitura(2)
saiLeitura(2, 0)
entraLeitura(2)
saiLeitura(2, 0)
entraLeitura(2)
saiLeitura(2, 0)
entraLeitura(2)
saiLeitura(2, 0)
entraLeitura(3)
saiLeitura(3, 0)
```

Figuras 23 e 24. Exemplo de log de saída do programa concorrente.

```
Execucao finalizada sem erros!
```

Figura 25. Exemplo de saída do programa auxiliar para o arquivo de log acima.

4. Conclusão

Como projeto, o trabalho proposto mostrou-se desafiador, porém, os conhecimentos da disciplina e resultantes do primeiro trabalho foram essenciais para que este trabalho ocorresse de forma mais fácil e com menos problemas. Portanto, podemos dizer que os trabalhos alcançam o objetivo de apurar o conhecimento do aluno.

5.1 Problemas e dificuldades encontrados

A primeira grande dificuldade encontrada foi o entendimento do enunciado proposto e do papel que o programa auxiliar deveria desempenhar. Os membros do grupo acharam difícil traduzir a verificação de corretude do programa concorrente para um programa sequencial.

Além disso, houve dificuldade na escrita do log de saída para o programa concorrente, visto que as escritas no arquivo deveriam ser bem colocadas para que a verificação ocorresse da forma correta.

Não houveram muitas dificuldades na concepção da solução concorrente para o problema, por ser um problema já visto muitas vezes em sala de aula. Sendo as dificuldades encontradas, e já citadas, referentes a geração do arquivo de log de saída e ao programa auxiliar.

5.2 Alcance dos objetivos

Como as dificuldades encontradas se resumem ao programa auxiliar e arquivos de saída, podemos dizer que o objetivo de colocar em prática os conceitos aprendidos em sala de aula foi alcançado.

Quando à solução do problema proposto, temos o comportamento que é esperado. Não verificamos nenhum erro de threads que são bloqueadas quando não deveriam ser e nas leituras e escritas no recurso compartilhado. Já em respeito à inanição de threads, não encontramos este problema nos testes realizados. Dessa forma, o grupo se sente confiante de que a ausência de inanição está garantida e este objetivo foi alcançado.