



UNIVERSIDADE FEDERAL DE SANTA MARIA
SISTEMAS OPERACIONAIS
SISTEMAS DE INFORMAÇÃO

RHAUANI WEBER AITA FAZUL

ANÁLISE DE PRIORIDADES E CASOS DE POSTERGAÇÃO
INDEFINIDA NAS SOLUÇÕES PARA O PROBLEMA DOS
LEITORES-ESCRITORES

Santa Maria, RS, Brasil
Junho de 2017

SUMÁRIO

1	INTRODUÇÃO	03
2	PRIMEIRA SOLUÇÃO – FAVORECENDO LEITORES	03
2.1	ANÁLISE DAS PRIORIDADES	04
2.2	POSTERGAÇÃO INDEFINIDA DOS ESCRITORES	05
3	SEGUNDA SOLUÇÃO – FAVORECENDO ESCRITORES	06
3.1	ANÁLISE DAS PRIORIDADES	06
3.2	POSTERGAÇÃO INDEFINIDA DOS LEITORES	07
4	TERCEIRA SOLUÇÃO – IMPLEMENTAÇÃO JUSTA	08
4.1	CONDIÇÃO PARA PRIORIDADES IGUAIS	09

1. INTRODUÇÃO

O presente relatório foi escrito com intuito de analisar os algoritmos implementados na resolução do problema dos leitores-escretores (*readers-writers problem*), fazendo uso de semáforos para controle de sincronização entre as *threads*.

A análise consiste em testes de prioridade e de postergação indefinida (*starvation*) nas três soluções do problema, sendo estas: **i) Prioridade dos leitores** (Também conhecido por First readers-writers problem, readers-preference, starving writers, favoring readers, reader-priority), **ii) Prioridade dos escritores** (Second readers-writers problem, writers-preference, starving readers, favoring writers, writer-priority) e **iii) Prioridades iguais** (Third readers-writers problem, no-starvation solution, fair solution). Foram consideradas três threads leitoras e duas threads escritoras para todos os testes realizados.

Também será feita uma breve explicação a respeito das implementações realizadas, abordando suas seções críticas e seus respectivos semáforos utilizados para sincronização.

2. PRIMEIRA SOLUÇÃO – FAVORECENDO LEITORES

Nesta solução, apresentada pela Figura 1, diferentes *threads* leitoras podem utilizar o mesmo recurso (base de dados) simultaneamente, ao passo que *threads* escritoras devem requisitar o recurso individualmente.

```
74 void *leitor(void *arg) {
75     int id = *(int*) arg;
76     while (TRUE) {
77         // Protocolo Entrada
78         sem_wait(&mutex);
79         /* Seção crítica */
80         if ( (rc += 1) == 1 )
81             sem_wait(&recurso);
82         sem_post(&mutex);
83
84         le_base(id);
85
86         // Protocolo saída
87         sem_wait(&mutex);
88         if ( (rc -= 1) == 0 )
89             sem_post(&recurso);
90         sem_post(&mutex);
91
92         usa_dados_lidos(id);
93     }
94 }
```

```
sem_t recurso // acesso a base de dados
sem_t mutex   // controle rc (read_count)
int rc        // qtd. processos lendo ou querendo ler

97 void *escritor(void *arg) {
98     int id = *(int*) arg;
99     while (TRUE) {
100         pensa_dados_escrita(id);
101         sem_wait(&recurso);
102         escreve_base(id);
103         sem_post(&recurso);
104     }
105 }
```


Figure 1 - Primeiro problema dos leitores-escretores

Um ponto a ser ressaltado é que mesmo que um leitor possa ler sempre que desejar e não houver escritor já realizando escrita (pode haver escritor esperando), ainda é preciso exclusão mútua entre leitores na execução das sessões de entrada e saída, por isso se faz necessário o uso do semáforo ‘mutex’, que controla o acesso a variável ‘rc’ (contador de leitores).

Isso irá evitar condições de corrida onde dois leitores incrementam o valor da variável contadora ao mesmo tempo e, em seguida, a tentativa de ambos ao trancar (na entrada) ou liberar (na saída) o recurso, causando o bloqueio de um dos leitores.

2.1 ANÁLISE DAS PRIORIDADES

A partir do momento em que o primeiro leitor conseguir acesso ao recurso, os leitores consecutivos não irão precisar requisitá-lo novamente, isso irá evitar o rebloqueio e permitir mais de um leitor ao mesmo tempo na base de dados (seção crítica). Portanto, depois do recurso estar em posse dos leitores, nenhum escritor poderá usá-lo antes de sua liberação, que somente será realizada quando a variável contadora de leitores ‘rc’, for zero, ou seja, nenhum leitor está dentro da seção crítica ou deseja entrar na mesma.

A terminal window titled 'rhau@rhau-PC: ~/Desktop/Trab3SO/codigos_teste' displays a sequence of status messages. The messages show three readers (Leitor 1, 2, 3) and one writer (Escritor 1). The sequence starts with 'Escritor 1 esta pensando...', followed by readers entering and using the resource. After several cycles of readers, the writer enters with 'Escritor 1 esta escrevendo'. The terminal ends with a '^C' signal and the prompt 'rhau@rhau-PC:~/Desktop/Trab3SO/codigos_teste\$'.

```
rhau@rhau-PC: ~/Desktop/Trab3SO/codigos_teste
Escritor 1 esta pensando...
Leitor 2 esta lendo dados
Leitor 3 esta lendo dados
Leitor 1 esta lendo dados
Leitor 2 esta usando os dados...
Leitor 3 esta usando os dados...
Leitor 1 esta usando os dados...
Leitor 2 esta lendo dados
Leitor 3 esta lendo dados
Leitor 1 esta lendo dados
Leitor 2 esta usando os dados...
Leitor 3 esta usando os dados...
Leitor 1 esta usando os dados...
Leitor 3 esta lendo dados
Leitor 2 esta lendo dados
Leitor 3 esta usando os dados...
Leitor 1 esta lendo dados
Leitor 3 esta lendo dados
Leitor 2 esta usando os dados...
Leitor 3 esta usando os dados...
Leitor 1 esta usando os dados...
Escritor 1 esta escrevendo
^C
rhau@rhau-PC:~/Desktop/Trab3SO/codigos_teste$
```

Figure 2 - Prioridade dos leitores em relação aos escritores

Um exemplo da prioridade explícita dos leitores pode ser vista na figura a cima, onde mesmo após esperar pelos leitores, que já estavam utilizando o recurso, acabarem sua leitura, o

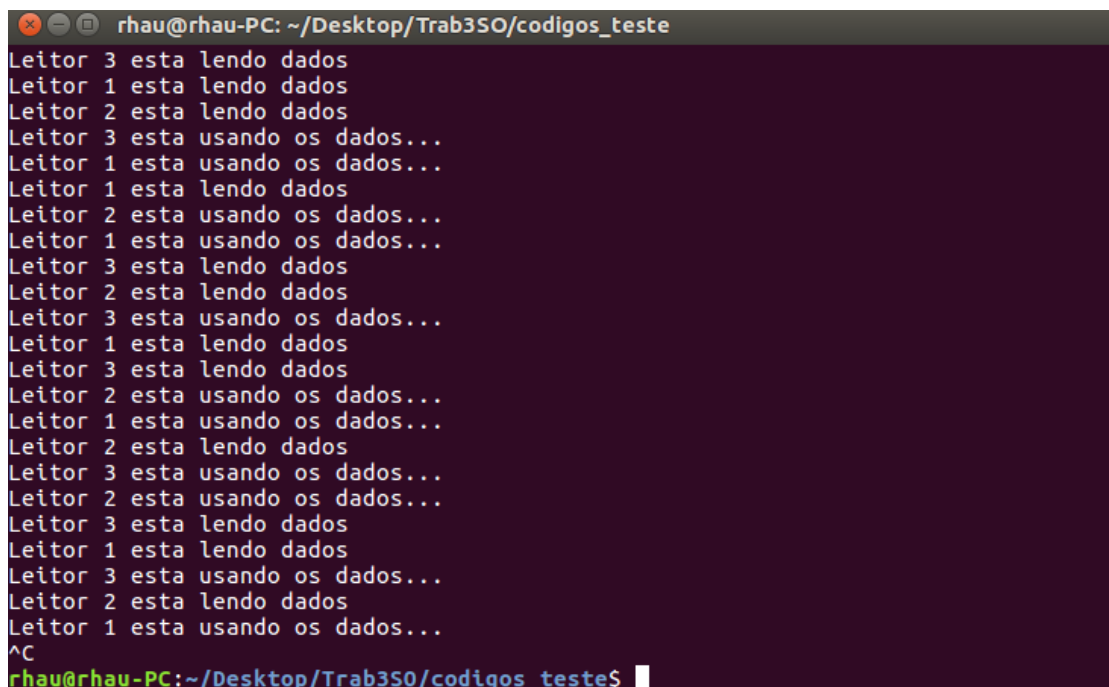
escritor teve de esperar por novos leitores que chegaram após ele próprio. Esse tipo de situação acaba resultando na *starvation* dos escritores.

Vale ressaltar que existe um momento em que as prioridades dos leitores e escritores serão iguais. Isto se dá quando um escritor terminar sua escrita (supondo que ele conseguiu escrever) e existem tanto leitores quanto escritores esperando para utilizar o recurso. Não existe prioridade dos leitores nessa situação: o próximo a executar dependerá do escalonador, que geralmente utiliza o método *first-in first-out*, ou seja, o primeiro a ter solicitado o semáforo do recurso, ganhará o acesso.

2.2 POSTERGAÇÃO INDEFINIDA DOS ESCRITORES

Como já citado anteriormente, apenas a primeira thread leitora precisa requisitar o recurso, logo, mesmo que exista um escritor esperando, se chegarem novos leitores, o que é muito provável, haverá o incremento da variável 'rc', o que fará com que a thread leitora em execução não libere o recurso e o escritor espere indefinidamente.

Portanto, situações como a da figura abaixo são comuns de serem encontradas.



```
rhau@rhau-PC: ~/Desktop/Trab350/codigos_teste
Leitor 3 esta lendo dados
Leitor 1 esta lendo dados
Leitor 2 esta lendo dados
Leitor 3 esta usando os dados...
Leitor 1 esta usando os dados...
Leitor 1 esta lendo dados
Leitor 2 esta usando os dados...
Leitor 1 esta usando os dados...
Leitor 3 esta lendo dados
Leitor 2 esta lendo dados
Leitor 3 esta usando os dados...
Leitor 1 esta lendo dados
Leitor 3 esta lendo dados
Leitor 2 esta usando os dados...
Leitor 1 esta usando os dados...
Leitor 2 esta lendo dados
Leitor 3 esta usando os dados...
Leitor 2 esta usando os dados...
Leitor 3 esta lendo dados
Leitor 1 esta lendo dados
Leitor 3 esta usando os dados...
Leitor 2 esta lendo dados
Leitor 1 esta usando os dados...
^C
rhau@rhau-PC:~/Desktop/Trab350/codigos_teste$
```

Figure 3 - Apenas leitores executam

3. SEGUNDA SOLUÇÃO – FAVORECENDO ESCRITORES

Nesta solução, apresentada pela Figura 4, quando uma thread escritora desejar escrever, nenhuma thread leitora poderá realizar sua leitura até que o escritor tenha sido atendido.

```
89 void *leitor(void *arg) {
90     int id = *(int*) arg;
91     while (TRUE) {
92         // Protocolo de entrada
93         sem_wait(&mutex_prioridade);
94         sem_wait(&tentativa_leitura);
95         sem_wait(&mutex_rc);
96         if ( (rc += 1) == 1 )
97             sem_wait(&recurso);
98         sem_post(&mutex_rc);
99         sem_post(&tentativa_leitura);
100        sem_post(&mutex_prioridade);
101
102        le_base(id);
103
104        // Protocolo de saida
105        sem_wait(&mutex_rc);
106        if ( (rc -= 1) == 0 )
107            sem_post(&recurso);
108        sem_post(&mutex_rc);
109        usa_dados_lidos(id);
110    }
111 }

113 void *escritor(void *arg) {
114     int id = *(int*) arg;
115     while (TRUE) {
116         pensa_dados_escrita(id);
117         // Protocolo de entrada
118         sem_wait(&mutex_wc);
119         if ( (wc += 1) == 1 )
120             sem_wait(&tentativa_leitura);
121         sem_post(&mutex_wc);
122
123         sem_wait(&recurso);
124         escreve_base(id);
125         sem_post(&recurso);
126
127         // Protocolo de saida
128         sem_wait(&mutex_wc);
129         if ( (wc -= 1) == 0 )
130             sem_post(&tentativa_leitura);
131         sem_post(&mutex_wc);
132     }
133 }
```

sem_t recurso // controla acesso base de dados
sem_t mutex_prioridade, tentativa_leitura
sem_t mutex_rc, mutex_wc // controla acesso 'rc', 'wc'
int rc, wc // read_count, write_count

Figure 4 - Segundo problema dos leitores-escretores

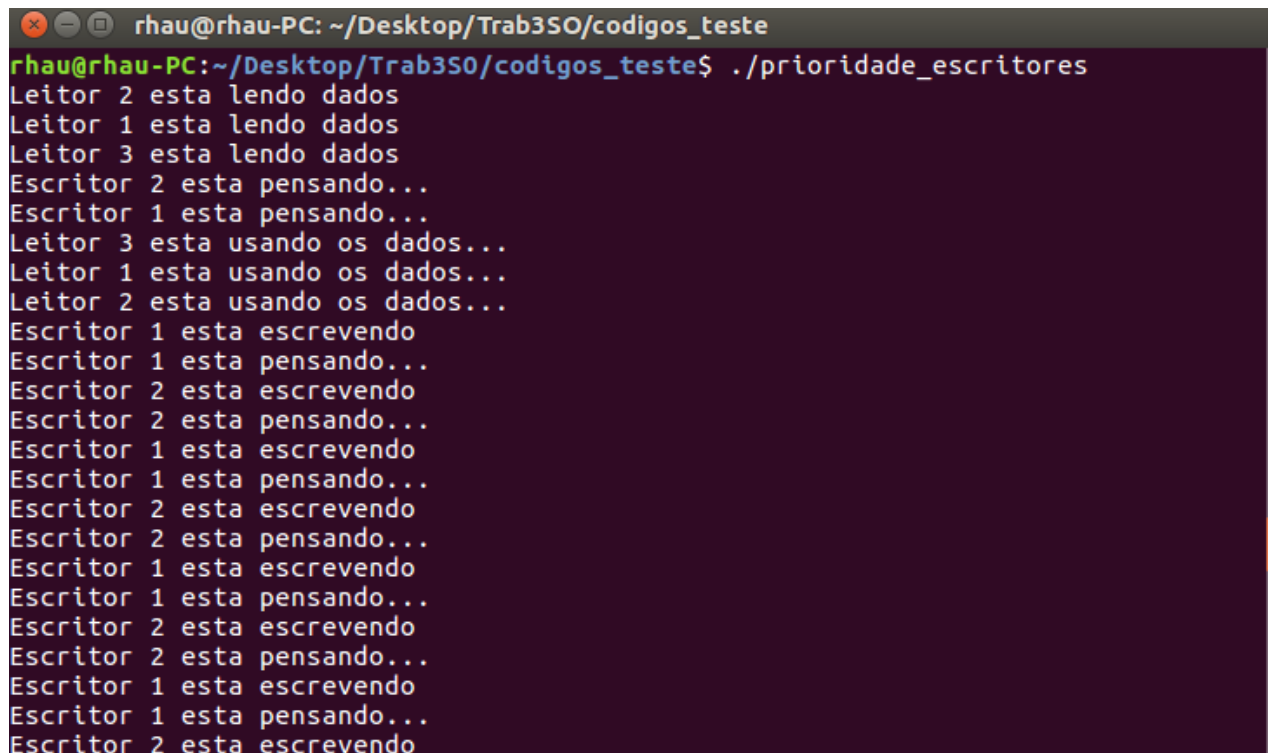
Assim como na primeira solução, o acesso a variável contadora exige exclusão mútua, a diferença é que agora se faz necessário o uso de um contador para os escritores também, assim como o semáforo ‘mutex_wc’ para evitar as condições de corrida explicadas na solução anterior.

3.1 ANÁLISE DAS PRIORIDADES

Como agora se deseja priorizar os escritores, cada leitor deverá requisitar o semáforo ‘tentativa_leitura’ individualmente, ao passo que o primeiro escritor irá bloquear este semáforo para si próprio e para todos os escritores consecutivos e somente o último escritor irá abrir o semáforo para os leitores tentarem realizar a leitura.

Caso não existam escritores tentando obter o recurso, então os leitores poderão realizar a leitura. Quando um escritor chegar, ele obtém o recurso o mais cedo possível, ou seja, espera apenas pelo leitor atual acabar sua leitura, pois de outra maneira ele teria de esperar pela fila de

leitores inteira. A figura abaixo demonstra a prioridade de escritores em relação a leitores.



```
rhau@rhau-PC: ~/Desktop/Trab3SO/codigos_teste
rhau@rhau-PC:~/Desktop/Trab3SO/codigos_teste$ ./prioridade_escritores
Leitor 2 esta lendo dados
Leitor 1 esta lendo dados
Leitor 3 esta lendo dados
Escritor 2 esta pensando...
Escritor 1 esta pensando...
Leitor 3 esta usando os dados...
Leitor 1 esta usando os dados...
Leitor 2 esta usando os dados...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
```

Figure 5 - Escritores possuem a prioridade

Para garantir essa prioridade, o semáforo ‘mutex_prioridade’ é necessário, pois sem ele haveria a possibilidade de um escritor e um ou mais leitores estarem simultaneamente parados em *sem_wait(&tentativa_leitura)* esperando a sua liberação ser feita por um leitor. Quando isso acontecesse a prioridade do escritor não poderia ser garantida, ou seja, todos os leitores da fila poderiam executar antes dele. O ‘mutex_prioridade’ garante acesso exclusivo ao bloco de código de dentro do semáforo, logo, até ser feita a liberação do semáforo ‘tentativa_leitura’, nenhum outro leitor poderá ficar parado para competir com algum escritor por sua liberação, com isso o escritor que estiver esperando o sinal irá executar sem nenhuma concorrência.

3.2 POSTERGAÇÃO INDEFINIDA DOS LEITORES

De maneira análoga a primeira solução, os leitores precisam esperar até o último escritor terminar sua escrita e liberar o semáforo para tentarem realizar sua leitura.

```
rhau@rhau-PC: ~/Desktop/Trab3SO/codigos_teste
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Escritor 1 esta escrevendo
^C
rhau@rhau-PC:~/Desktop/Trab3SO/codigos_teste$
```

Figure 6 - Apenas escritores executam

Dessa maneira, utilizando como exemplo a figura a cima, apenas quando não houverem mais escritores escrevendo ou tentando escrever, os leitores poderão executar.

4. TERCEIRA SOLUÇÃO – IMPLEMENTAÇÃO JUSTA

Com a finalidade de evitar a postergação indefinida de leitores ou escritores vistas nas implementações anteriores, a terceira solução visa executar ambos os tipos de threads com as mesmas prioridades, eliminando o risco de *starvation*. Como consequência, poderá haver uma queda de desempenho em relação às demais soluções.

Leitores e escritores irão receber acesso ao recurso em função de sua ordem de chegada, se um escritor chegar enquanto um ou mais leitores estão acessando o recurso, ele terá esperar até estes leitores finalizarem sua leitura para, somente depois, realizar sua escrita. Se novos leitores chegarem nesse meio tempo, eles também terão de esperar a liberação do recurso.

A Figura 7 ilustra a justiça do algoritmo. Tanto leitores quanto escritores estão aptos a competir igualmente para executar.


```

rhau@rhau-PC: ~/Desktop/Trab3SO/codigos_teste
rhau@rhau-PC:~/Desktop/Trab3SO/codigos_teste$ ./prioridades_iguais
Leitor 2 esta lendo dados
Leitor 1 esta lendo dados
Leitor 3 esta lendo dados
Escritor 1 esta pensando...
Escritor 2 esta pensando...
Leitor 2 esta usando os dados...
Leitor 1 esta usando os dados...
Leitor 3 esta usando os dados...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Leitor 2 esta lendo dados
Leitor 1 esta lendo dados
Leitor 1 esta usando os dados...
Leitor 2 esta usando os dados...
Escritor 2 esta escrevendo
Escritor 2 esta pensando...
Leitor 3 esta lendo dados
Leitor 3 esta usando os dados...
Escritor 1 esta escrevendo
Escritor 1 esta pensando...
Leitor 2 esta lendo dados
Leitor 1 esta lendo dados
Leitor 2 esta usando os dados...

```

Figure 7 - Leitores e escritores executam

4.1 CONDIÇÃO PARA PRIORIDADES IGUAIS

Para permitir esta "fila justa" entre leitores e escritores e consequentemente prevenir a postergação indefinida, é necessário usar o semáforo 'fila', que pode ser visto na figura a baixo, para enfileirar as requisições por ordem de chegada. Qualquer thread que desejar o recurso deverá bloquear este semáforo e liberá-lo, assim que ganhar o acesso ao recurso em si.

<pre> 74 void *leitor(void *arg) { 75 int id = *(int*) arg; 76 while (TRUE) { 77 // Protocolo Entrada 78 sem_wait(&fila); 79 sem_wait(&mutex_rc); 80 if ((rc += 1) == 1) 81 sem_wait(&recurso); 82 83 sem_post(&fila); 84 sem_post(&mutex_rc); 85 86 le_base(id); 87 88 // Protocolo saida 89 sem_wait(&mutex_rc); 90 if ((rc -= 1) == 0) 91 sem_post(&recurso); 92 93 sem_post(&mutex_rc); 94 usa_dados_lidos(id); 95 } 96 } </pre>	<pre> 98 void *escritor(void *arg) { 99 int id = *(int*) arg; 100 while (TRUE) { 101 pensa_dados_escrita(id); 102 sem_wait(&fila); 103 sem_wait(&recurso); 104 sem_post(&fila); 105 106 escreve_base(id); 107 108 sem_post(&recurso); 109 } 110 } </pre>
--	--

sem_t fila	// turno, ordem de chegada
sem_t recurso	// controla acesso base de dados
sem_t mutex_rc	// controla acesso 'rc'
int rc	// contador de leitores

Figure 8 - Terceiro problema dos leitores-escretores

O semáforo deverá preservar a ordem *first-in first-out* (FIFO) ao bloquear e liberar as threads, de outra maneira, poderá ocorrer de, por exemplo, uma thread escritora bloqueada permanecer em bloqueio indefinidamente por conta de um ciclo de outras threads, escritoras ou não, que conseguiriam decrementar o semáforo antes dela própria.