Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Aula Prática 05 - *Locks* (Complementar)

## Identificação da Disciplina

| Código da Disciplina | Nome da Disciplina | Turma | Professor | Período |
|---|---|---|---|---|
| 117935 | Programação Concorrente | A | Jeremias Moreira Gomes | 2019/2 |

## Objetivo da Aula Prática

O objetivo desta aula prática é apresentar técnicas básicas de comunicação entre processos/*threads* com sincronização por meio primitivas envolvendo bloqueio de processos.

## Detalhes Acerca das Aulas Práticas

### 0.1. Restrições deste Trabalho Prático

Durante a escrita de quaisquer códigos, tentar não utilizar bibliotecas exóticas que não sejam as disponíveis no UNIX como biblioteca padrão. Se o fizer (pois ajuda na realização de testes, por exemplo), lembrar que no momento da correção o sistema padrão a ser utilizado é similar a uma instalação padrão das máquinas do LINF, e nada além do código será instalado apenas para uma correção individual. Então o uso de uma `<gtest.h>`, por exemplo, pode acabar fazendo com que o aluno perca pontos, porque seu código não pôde ser testado, em virtude da falta desta no computador de testes.

O atraso da entrega da atividade, incorre no decondo de $0.5$ pontos a cada $60$ minutos de atraso, em relação ao prazo de entrega da atividade e da nota final do aluno nesta atividade.

Esta atividade pode ser feita em dupla.

### 0.2. Por onde entregar os exercícios das aulas práticas?

As atividades envolvendo as aulas práticas deverão ser entregues via Aprender (https://aprender.ead.unb.br), na disciplina de Programação Concorrente. As informações para ingresso são as seguintes:

- URL: https://aprender.ead.unb.br/course/view.php?id=6775
- Chave de Acesso: `s3nhaD3ss3semestre201902`

Após o ingresso na disciplina, haverá uma atividade chamdada “Aula Prática 05 - *Locks* (Complementar)”, para submissão dos exercícios.

### 0.3. O que deverá ser entregue, referente as aulas práticas?

Deverão ser entregues respostas referentes a todos os tópicos das **Seções de Atividades** ao longo deste documento. Essas atividades (dessa aula prática) estão divididas em duas categorias:

- Questionários (pergunta e resposta).
- Elaboração de Códigos.

Todos os dois tipos de atividades deverão ser entregues em um documento único contendo todas as respostas. Esse documento deverá ter identificação do aluno (nome e matrícula), identificação da disciplina, identificação da aula prática e as respostas identificadas de maneira igual as numerações em que aparecem neste documento. Para auxiliar na elaboração desse documento, pode-se utilizar esse documento (clique aqui) de referência.

Além disso, as questões de elaboração de código também deverão ser entregues em arquivos (.c) separados, sendo um arquivo para cada código elaborado. O início desse código deverá vir com comentários fazendo referência ao autor do código, nome do arquivo e a identificação da atividade, da seguinte forma:

```
// autor: Jeremias Moreira Gomes
// arquivo: exemplo-arquivo.c
// atividade: 0.0.0

#include <stdio.h>

int main()
{
    printf("\n");
    return 0;
}
```

Assim, os códigos elaborados irão estar em arquivos separados e no relatório.

### 0.4. Como entregar as atividades das aulas práticas?

A submissão das atividades deverá ser feita em um arquivo único comprimido no tipo zip (`Zip archive data, at least v1.0 to extract`) contendo um diretório com o relatório e os códigos elaborados durante a atividade. Além disso, para garantir a integridade do conteúdo entregue, o nome do arquivo comprimido deverá possuir duas informações (além da extensão `.zip`):

- A matrícula do aluno (ou dos alunos separadas por hífen, em caso de dupla).
- O *hash* md5 do arquivo `.zip`.

Para gerar o md5 do arquivo comprimido, utilize o comando `md5sum` do Linux e em seguida faça o renomeamento utilizando o *hash* coletado. Exemplo:

```
[6189] j3r3mias@tardis:aula-01-processos|master > zip -r aaa.zip atividade-01/
  adding: atividade-01/ (stored 0%)
  adding: atividade-01/relatorio.docx (stored 0%)
  adding: atividade-01/01-hello-3-fork.c (deflated 34%)
  adding: atividade-01/03-exemplos.c (deflated 47%)
  adding: atividade-01/02-arvore.c (deflated 35%)
  adding: atividade-01/02-pid_t.c (deflated 39%)
  adding: atividade-01/01-hello-fork.c (deflated 21%)
  adding: atividade-01/03-processos-e-ordens.c (deflated 55%)
[6190] j3r3mias@tardis:aula-01-processos|master > ls -lha aaa.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 aaa.zip
[6191] j3r3mias@tardis:aula-01-processos|master > md5sum aaa.zip
fe36b6799eae103a464cbc4857fce404  aaa.zip
[6192] j3r3mias@tardis:aula-01-processos|master > mv aaa.zip 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6193] j3r3mias@tardis:aula-01-processos|master > ls -lha 160068444-fe36b6799eae103a464cbc4857fce404.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6194] j3r3mias@tardis:aula-01-processos|master > md5sum 160068444-fe36b6799eae103a464cbc4857fce404.zip
fe36b6799eae103a464cbc4857fce404  160068444-fe36b6799eae103a464cbc4857fce404.zip
[6195] j3r3mias@tardis:aula-01-processos|master >
```

# 1. Problema dos Leitores e Escritores

Um problema muito famoso na área de Programação Concorrente é o problema dos leitores e escritores. A principal diferença deste para outros problemas já apresentados até agora está na diferença entre as funções de cada um dos processos ou *threads* dentro do contexto do problema. O problema é o seguinte: modelar o acesso a uma base de dados, onde algumas *threads* estão lendo os dados da região crítica e, por consequência, somente querendo obter a informação da região crítica, chamadas de *threads* leitoras, e outras *threads* estão tentando alterar a informação da região crítica, chamadas de *threads* escritoras.

A partir do enunciado, é possível perceber que quando há somente *threads* leitoras envolvidas no "processo", estas podem ser liberadas para acessarem ao mesmo tempo o conteúdo do banco de dados, enquanto apenas uma *thread* escritora pode modificar o conteúdo do banco por vez. Além disso, deve-se lembrar que enquanto a *thread* escritora estiver escrevendo, nem as *threads* leitoras podem ler o conteúdo ao mesmo tempo. Dessa forma, existem situações onde podem-se ter *threads* ao mesmo tempo dentro da região crítica (leitoras) e situações onde pode-se ter apenas uma única *thread* (escritora) dentro da região crítica.

## 1.1. Atividades

### 1.1.1. Implementação do Problema dos Leitores e Escritores

Sua tarefa neste tópico é implementar o problema dos Leitores e Escritores. Para a implementação, deve-se seguir os seguintes tópicos:

- O banco de dados será uma cadeia de caracteres com 10000 posições, inicialmente com o caracter `'A'`.
- Deverão ser criadas 10 *threads* leitoras e 3 *threads* escritoras.
- Cada *thread* escritora deverá percorrer um laço por 1000 (mil) iterações.
- A cada iteração, uma *thread* escritora deverá obter o acesso exclusivo a região crítica, por meio de um semáforo ou mutex, e concatenar o seu identificador ao final do banco de dados.
- Os identificadores de cada *thread* escritora deverão ir de 0 a 9.
- Em seguida a escrita, a *thread* escritora deverá liberar o acesso a região crítica e dormir por um valor pseudo-aleatório entre 0 a 2 segundos.
- *Threads* leitoras deverão funcionar da seguinte maneira:
    1. Leitor dever ter um laço de repetição contínuo para leitura.
    2. Deverá existir uma região crítica para leitura (1).
    3. Deverá existir um controle de leitores dentro da região crítica (um contador).
    4. Se o leitor for o primeiro a entrar na região crítica (1), este deverá obter o acesso exclusivo da região crítica que o escritor está utilizando.
    5. O leitor deverá liberar o acesso a sua região crítica (1).
    6. O leitor realiza a leitura ao banco banco de dados (impressão do conteúdo da variável (não se preocupe com comportamentos estranhos da função `printf` por não estar protegida aqui)).
    7. O leitor reobtém a trava de acesso a sua região crítica de leitura (2).
    8. O leitor sinaliza que não está mais realizando leitura (modificando o contador).
    9. Se o leitor, for o único leitor realizando leitura, este deverá liberar o acesso para escrita da região crítica que o escritor está utilizando.
    10. O leitor libera o acesso a sua região crítica (2).
    11. O leitor dorme um valor pseudo-aleatório entre 0 e 3 segundos.

**1.2. A partir da implementação apresentada no tópico anterior, se fossem removidos os *sleeps* das *threads* leituras, existe a chance de um problema acontecer. Que problema é esse e como ele ocorre?**

## 2. *Locks* Recursivos

A partir do tópico anterior, é possível perceber que a utilização de um mesmo *lock* para diferentes funções é um cenário plenamente possível em implementação de problemas de Programação Concorrente. Um problema a que se pode chegar, a partir do manuseio de travas dessas forma, é a possibilidade de impasses devido ao uso de recursividade em cima das chamadas dos *locks*. Por recursividade nas chamadas, entenda-se o caso onde uma *thread* que já tem o acesso exclusivo a uma região crítica tenta obter essa trava novamente. Segue um exemplo simples.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex;

void bar()
{
    printf("Tentando pegar o lock de novo.\n");
    pthread_mutex_lock(&mutex);
    printf("Estou com duplo acesso?\n");
    pthread_mutex_unlock(&mutex);
}

void *foo(void *empty)
{
    pthread_mutex_lock(&mutex);
    printf("Acesso a região crítica.\n");
    bar();
    pthread_mutex_unlock(&mutex);
}

int main()
{
    pthread_t t;

    pthread_mutex_init(&mutex, 0);
    pthread_create(&t, NULL, foo, NULL);
    pthread_join(t, NULL);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

No trecho de código anterior, é possível perceber que, logo após a *thread* obter o *lock* na função `foo`, ela tenta obter o mesmo *lock* ao chamar a função `bar`, gerando um impasse o qual ela fica aguardando por um recurso que ela mesma já possui, assim nunca continuando a sua execução. Para evitar esse problema algumas abordagens podem ser utilizadas. A primeira delas consiste em refatorar o código de modo a uma função interna nunca chamar a região crítica de uma função de nível superior. A segunda abordagem, consiste no uso de mutexes com propriedade de usos recursivas a qual, de maneira análoga a semáforos, pode ser chamada mais de uma vez por uma mesma *thread* e não irá ocasionar o seu bloqueio, pois funciona similarmente a um contador (no caso de uma mesma *thread*). Por esse comportamento, mutexes com características recursivas devem ser liberados, pela mesma *thread*, tantas vezes quantas forem chamadas, para que outra *thread* consiga acesso a região crítica.

## 2.1. Atividade

### 2.1.1. Conserte o código anterior sem refatorar as funções.

O objetivo desta tarefa é modificar as propriedades do mutex, para consertar o código anterior sem a necessidade de modificar as funções `foo` e `bar`. Dessa forma, deve-se utilizar algumas das diferentes propriedades dos mutexes, para que o mesmo passe a funcionar com característica recursiva. Para isso, consulte a parte de configurações das propriedades dos mutexes (linux.die.net) ou a bíblia, para aprender como utilizar.