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

# Aula Prática 06 - *Deadlocks*

## 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 06 - *Deadlocks*", 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. Modelagem de Processos Para Detecção de *Deadlocks*

Em 1972, Richard C. Holt apresentou uma modelagem dessas quatro condições para a detecção de *deadlocks*, por meio de grafos dirigidos. Dessa forma, é possível realizar a identificação de ocorrências de *deadlocks* a partir da construção do modelo, dada a sequência de eventos no ambiente.

### 1.1. Atividades

#### 1.1.1. A partir do extrato código abaixo, e de uma possível sequência de eventos, represente cada estado do sistema após a ocorrência de cada evento.

```
// Processo A
requisita(K)
requisita(L)
libera(L)
libera(K)
```

```
// Processo B
requisita(L)
requisita(M)
libera(M)
libera(L)
```

```
// Processo C
requisita(L)
requisita(K)
requisita(M)
libera(M)
libera(K)
libera(L)
```

```
// Processo D
requisita(M)
requisita(N)
libera(N)
libera(M)
```

Sequência de eventos:

```
A requisita K
A requisita L
B requisita L
B requisita M
C requisita L
A libera L
A libera K
C requisita K
```

#### 1.1.2. Utilizando o extrato de código anterior, identifique se é possível que ocorra algum *deadlock*. Se for possível, descreva a sequência de eventos para que este ocorra, bem como a sua representação gráfica.

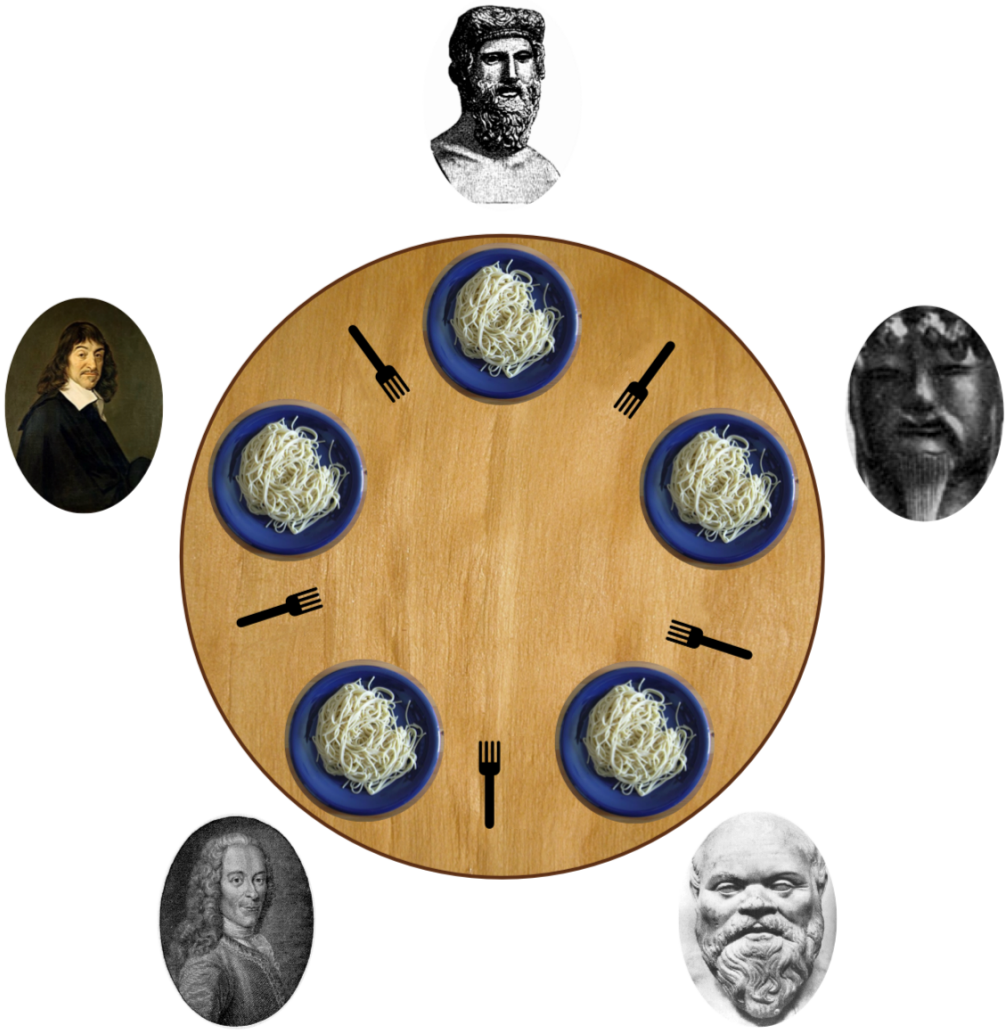
```
A requisita K
C requisita L
A requisita L
C requisita K
```

## 2. O Jantar dos Filósofos

O Jantar dos Filósofos é um problema que foi proposto por Dijkstra em 1965 como um problema de sincronização. Por ser um problema famoso na área de Sistemas Operacionais e Programação Concorrente, todos os algoritmos propostos como soluções de sincronização acabaram sendo relacionados ou testados contra o problema do Jantar dos Filósofos.

**Definição do Problema:**

Cinco filósofos estão sentados em uma mesa redonda para jantar. Cada filósofo tem um prato com espaguete à sua frente. Cada prato possui um garfo para pegar o espaguete. O espaguete está muito escorregadio e, para que um filósofo consiga comer, será necessário utilizar dois garfos. O problema é que só existe um garfo entre cada prato. Além disso, cada filósofo alterna entre duas tarefas: comer e pensar. Quando um filósofo fica com fome, ele tenta pegar os garfos à sua esquerda e à sua direita; um de cada vez, independente da ordem. Caso ele consiga pegar dois garfos, ele come durante um determinado tempo e depois recoloca os garfos na mesa. Em seguida ele volta a pensar.



O problema do Jantar dos Filósofos consiste em propor um algoritmo que implemente cada filósofo de modo que ele execute as tarefas de comer e pensar sem nunca ficar travado.

Apesar de parecer um problema simples, alguns desafios podem ser enfrentados na solução desse problema como *deadlocks*, quando todos os filósofos ficam com fome e todos pegam cada um garfo, e até *starvation*, onde algum filósofo nunca consegue pegar dois garfos para conseguir comer.

### 2.1. Atividade

#### 2.1.1. Implementar uma Solução para o Jantar dos Filósofos

Sua solução deverá funcionar da seguinte maneira:

- Cada filósofo será representado por uma *thread*.
- Continuamente, cada filósofo deverá Pensar (dormir por um segundo), pegar garfo e soltar garfo.
- Um filósofo poderá encontrar-se em um de três estados: PENSANDO, COMENDO e COM FOME.
- Quando o filósofo pegar um garfo, indica-se que este está com fome.
- Ao pegar o garfo, este deverá verificar se os garfos da esquerda e da direita, um por vez, em qualquer ordem, estão disponíveis. Uma dica para essa verificação, é olhar o próprio estado dos filósofos próximos e se ambos estiverem pensando ou com fome, este é um indício que os garfos ainda estão disponíveis. Assim, o filósofo passa para o estado de comendo e dorme por dois segundos.
- Ao soltar o garfo, o filósofo passa para o estado de pensando, indicando que este liberou o (s) garfo (s).

Dica: utilize contadores, para confirmar que todos os filósofos estão conseguindo comer (problema de *starvation*).

## 3. *Debbuging* de Aplicações com Múltiplas *Threads*

Por diversas vezes, é necessário "debuggar" códigos diversos. Com aplicações envolvendo múltiplas linhas de execução não é diferente. Assim, é possível verificar via `gdb` o estado de execução de diversas *threads* ao mesmo tempo.

Após iniciar um programa utilizando o `gdb` e este atingir um ponto de parada (*breakpoint*), o comando `info threads`, irá gerar informações a respeito das múltiplas linhas de execução do programa.

Considere o programa abaixo:

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

void *tarefa(void *id)
{
    long int tid = (long int )id;
    printf("Olá, eu sou %ld\n", tid);
}

int main()
{
    const int NUMTHREADS = 5;
    pthread_t threads[NUMTHREADS];

    for (long int i = 0; i < NUMTHREADS; i++) {
```

```
        pthread_create(&threads[i], NULL, tarefa, (void *) i);
    }

    for (long int i = 0; i < NUMTHREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

Essa é a saída do comando, ao colocar uma parada após a criação das *threads*.

```
gdb-peda$ info threads
  Id   Target Id         Frame
  1    Thread 0x7ffff7fbe740 (LWP 20093) "prog" clone () at ../sysdeps/unix/sy
* 2    Thread 0x7ffff77c4700 (LWP 20099) "prog" 0x000055555555477a in tarefa
  3    Thread 0x7ffff6fc3700 (LWP 20100) "prog" 0x000055555555477a in tarefa
  4    Thread 0x7ffff67c2700 (LWP 20101) "prog" clone () at ../sysdeps/unix/sy
```

Além disso, é possível trocar entre as *threads* para continuar a execução a partir de alguma delas, utilizando o comando `thread <id>`. Para outros exemplos, você pode consultar o manual da Red Hat. (link).

Por último, existem ferramentas que auxiliam o programador na localização de alguns *deadlocks* em seu programa. A ferramenta Valgrind, por exemplo, possui um complemento chamado Helgrind que permite localizar situações como tentativa de destravar mutexes que não foram travados, destruição inválida de mutexes, chamadas recursivas a *locks* que não possuem tal propriedade, etc.

## 3.1. Atividade

### 3.1.1. Localização de *Deadlocks*

Dado o código abaixo:

```
#include <pthread.h>

int var = 0;

void* contador ( void* arg ) {
   var++;
}

int main ( void ) {
   pthread_t t[10];
   for (int i = 0; i < 10; i++) {
       pthread_create(&t[i], NULL, contador, NULL);
   }
   var++; // A main também irá incrementar
```

```
    for (int i = 0; i < 10; i++) {
        pthread_join(t[i], NULL);
    }
    return 0;
}
```

Utilize a ferramenta Valgrind para detectar condições de corrida dentro do código. Exiba o relatório apresentado pela ferramenta, bem como uma extração dos pontos problemáticos no código.