EP 3 - Semáforo Binário

- Gabriel Hoffman - 10783250
- Guilherme Balog - 11270649

Solução

Nossa solução foi desenvolvida utilizando POSIX thread e uma de suas funções, a do **mutex**. Para isso criamos 2 threads que executavam a mesma função `void* runner(void *threadId)`. Nessa função, a região crítica exibe incrementa e exibe o *counter*, além de mostrar qual foi a *thread* responsável pela ação, assim podemos observar a alternância.

A seguir temos um trecho de uma execução:

```
...
Thread 2: Counter -> 130
Thread 2: Counter -> 131
Thread 2: Counter -> 132
Thread 2: Counter -> 133
Thread 2: Counter -> 134
Thread 1: Counter -> 135
Thread 1: Counter -> 136
Thread 1: Counter -> 137
Thread 1: Counter -> 138
Thread 1: Counter -> 139
Thread 2: Counter -> 140
Thread 1: Counter -> 141
Thread 2: Counter -> 142
Thread 1: Counter -> 143
Thread 2: Counter -> 144
Thread 1: Counter -> 145
Thread 2: Counter -> 146
Thread 1: Counter -> 147
...
```

Ambiente

O exercício programa foi desenvolvido no editor de texto Visual Studio Code, com a linguagem C, então compilamos com o `gcc`, em sua versão 9.3.0 e o sistema operacional foi o Ubuntu 20.04.

Condições

O problema da seção crítica tem as seguintes condições:

1. **Exclusão mútua.** Se o processo P_i está executando sua seção crítica, então nenhum outro processo pode executar sua seção crítica.

2. **Progresso.** Se nenhum processo está executando sua seção crítica e algum processo quer entrar em sua seção crítica, então somente aqueles processos que não estão executando suas seções remanescentes podem participar da decisão de qual entrará em sua seção crítica a seguir, e essa seleção não pode ser adiada indefinidamente.

3. **Espera limitada.** Há um limite, ou fronteira, para quantas vezes outros processos podem entrar em suas seções críticas após um processo ter feito uma solicitação para entrar em sua seção crítica e antes de essa solicitação ser atendida.

Implementando e analisando o código da solução, chegamos às seguintes justificativas sobre cada uma das condições:

1. As funções `pthread_mutex_lock` e `pthread_mutex_unlock` garantem que nenhuma outra *thread* entre na região do código que está entre a chamada de *lock* e *unlock*.

2. A condição de progresso é satisfeita pelo semáforo.

3. A espera não é limitada em nenhum momento, ou seja, pode acontecer da thread 2 entrar em execução apenas quando a thread 1 terminar.

Suposições

Acreditamos que devido a diversas condições do sistema o código pode nos retornar diferentes resultados quanto a alternância entre as threads. Isso porque tem vez que cada thread executa metade do processo e algumas vezes elas se alternam com mais frequência.

Por exemplo, tivemos resultados bem divididos, com este:

```
Thread 1: Counter -> 1
...
Thread 1: Counter -> 1000
Thread 2: Counter -> 1001
...
Thread 2: Counter -> 2000
```

Mas também resultados bem mais alternados, como este:

```
Thread 1: Counter -> 1
Thread 2: Counter -> 2
Thread 1: Counter -> 3
Thread 2: Counter -> 4
Thread 1: Counter -> 5
Thread 2: Counter -> 6
...
```