**Escola de Artes, Ciências e Humanidades** (EACH)

**Prática 03 - ACH2044 Sistemas Operacionais**
Relatório do experimento

**Nome:** Guilherme Cavalanti Gomes
**NºUSP:** 11844788

## O PROBLEMA

O objetivo desse EP, é implementar uma solução para a seção crítica utilizando semáforos binários. Será feito um programa que possui dois processos que compartilhem de uma mesma variável global, utilizando POSIX Threads em C.

Dessa forma, será usada uma seção crítica, em que os 2 processos se alternam para entrar e sair dela, sendo regulados por dois semáforos binários.

## IMPLEMENTANDO O ALGORITMO:

A lógica usada nessa solução é a do semáforo binário (mutex), ou seja, as threads irão esperar até um semáforo as deixarem entrar na seção crítica, e logo após irão dar um sinal para outro semáforo liberando ele.

Para isso serão utilizadas duas variáveis que controlam o acesso de cada um dos processos, através das funções wait e signal, criando um sistema de turnos.

A função wait faz com que o processo espere até o semáforo ter valor 1, então ele entra na seção crítica e após sair, zera o valor do semáforo de novo. Já a função signal incremente o valor do semáforo em 1, abrindo caminho para outro processo. A lógica é simples, cada processo espera pelo seu respectivo semáforo, e então dá um sinal para o outro semáforo, assim eles se alternam constantemente. Inicialmente um dos mutex tem valor 0 e o outro 1.

### O CÓDIGO EM C:

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

//contador
int count;

//semáforos
typedef int semaphore;
semaphore mutex0;

semaphore mutex1;
```

```c
//Função executada dentro da seção crítica
void secao_critica (int threadid){
    int id = (int) threadid;
    printf("(%d) Secao Critica, count = %i\n",id, count);
    count += 1;
}

//Função de espera
void secao_nao_critica (int threadid) {
    int id = (int) threadid;
    printf("(%d) Esperando\n",id);
}

//Wait e Signal
void wait (semaphore* S, int id){
    while(*S <= 0);
    secao_critica(id);
    *S = 0;
    return;
}
void signal (semaphore* S){
*S = 1;
}

// Processo 0
void *p0 (void *threadid){
    int id = (int) threadid;
    while (1) {
        // while(mutex != 0);
        wait(&mutex0, id);
        // secao_critica(id);
        signal(&mutex1);
        secao_nao_critica(id);
    }
}

// Processo 1
void *p1 (void *threadid){
    int id = (int) threadid;
    while (1) {
        // while(mutex != 0);
        wait(&mutex1, id);
```

```
        // secao_critica(id);
        signal(&mutex0);
        secao_nao_critica(id);
    }
}

// Main
void main(){
    mutex0 = 1;
    mutex1 = 0;
    count = 0;
    int num; //indetificador da thread
    pthread_t t1, t2;
    int numThread; // armazena o retorno da pthread_create
    num = 0;
    numThread = pthread_create(&t1, NULL, p0, (void *)num);
    num = 1;
    numThread = pthread_create(&t2, NULL, p1, (void *)num);
    pthread_join(t1,NULL); //Espera a t1 acabar
    pthread_join(t2,NULL); // Espera a t2 acabar
}
```

**Resultados:**

```
C:\Users\W10\Desktop\EP3-OAC>programa3.exe
(0) Secao Critica, count = 0
(0) Esperando
(1) Secao Critica, count = 1
(1) Esperando
(0) Secao Critica, count = 2
(0) Esperando
(1) Secao Critica, count = 3
(1) Esperando
(0) Secao Critica, count = 4
(1) Secao Critica, count = 5
(1) Esperando
(0) Esperando
(0) Secao Critica, count = 6
(0) Esperando
```

## Análise:

Como é possível observar, a solução é funcional, já que os 2 processos trabalham concorrentemente sem interferências e sem distorcer o valor da variável global ‘count’. Dito isso, é preciso analisar se ele atende as 3 condições necessárias para que haja exclusão mútua:

**1- Exclusão Mútua:** Atende, pois quando uma das threads está na seção crítica, a outra necessariamente vai estar esperando o sinal, e quando a thread acabar ele vai ter que esperar o sinal da próxima, que só vai chegar após o término da sua função. Assim elas nunca podem estar juntas na seção crítica.

**2-Progesso:** Ela é atendida pois, dentro das funções ‘p1’ e ‘p0’ que são executadas nas threads, há um loop infinito então elas nunca ficarão ociosas ao mesmo tempo, visto que o valor dos 2 mutex nunca vai ser 0, eles sempre estarão se alternando.

**3-Espera limitada:** Também é cumprida, pois, sempre ao fim da tarefa na seção crítica, há uma sinalização para a próxima thread, ou seja, sempre chegará a vez daquele processo não há como ele ficar na fila para sempre.

Logo essa solução é válida e atinge aos 3 pontos da exclusão mútua, um único ponto fraco seria que a ordem dos processos é definida por turnos, então se há 1 processo muito rápido e 1 muito lento, o rápido vai ter que esperar o lento terminar toda vez que seu turno acabar, gerando uma perda no desempenho se comparado à outras soluções.