FOLHA DE PROBLEMAS Nº 7

Mecanismos de Sincronização (Semáforos, *Mutexes* e *Condition Variables*), Memória Partilhada e Filas de Mensagens

---

1. – Sincronização entre *threads* acedendo a variáveis partilhadas, usando um *mutex* (1)

Considere o seguinte programa :

```
// PROGRAMA p01.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAXELEMS 10000000 // nr. max de posicoes
#define MAXTHREADS 100    // nr. max de threads
#define min(a, b) (a)<(b)?(a):(b)

int npos;
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;  // mutex p/a sec.critica
int buf[MAXELEMS], pos=0, val=0;                // variaveis partilhadas

void *fill(void *nr)
{
        while (1) {
                pthread_mutex_lock(&mut);
                if (pos >= npos) {
                        pthread_mutex_unlock(&mut);
                        return NULL;
                }
                buf[pos] = val;
                pos++; val++;
                pthread_mutex_unlock(&mut);
                *(int *)nr += 1;
        }
}

void *verify(void *arg)
{
        int k;
        for (k=0; k<npos; k++)
                if (buf[k] != k)   // detecta valores errados
                        printf("ERROR: buf[%d] = %d\n", k, buf[k]);
        return NULL;
}

int main(int argc, char *argv[])
{
        int k, nthr, count[MAXTHREADS];   // array para contagens
        pthread_t tidf[MAXTHREADS], tidv; // tids dos threads
        int total;

        if (argc != 3) {
                printf("Usage: %s <nr_pos> <nr_thrs>\n",argv[0]);
                return 1;
        }
```

```
        npos = min(atoi(argv[1]), MAXELEMS);    //no. efectivo de posicoes
        nthr = min(atoi(argv[2]), MAXTHREADS);  //no. efectivo de threads

        for (k=0; k<nthr; k++) {   // criacao das threads 'fill'
                count[k] = 0;
                pthread_create(&tidf[k], NULL, fill, &count[k]);
        }

        total=0;
        for (k=0; k<nthr; k++) {   //espera threads 'fill'
                pthread_join(tidf[k], NULL);
                printf("count[%d] = %d\n", k, count[k]);
                total += count[k];
        }
        printf("total count = %d\n",total);   // mostra total

        pthread_create(&tidv, NULL, verify, NULL);
        pthread_join(tidv, NULL);   // espera thread 'verify'

        return 0;
}
```

a) Analise e interprete o programa. Execute-o várias vezes e verifique que o resultado é o esperado.

b) Elimine as chamadas a `pthread_mutex_lock()` e `pthread_mutex_unlock()` na função `fill()` e execute o programa novamente. Verifique que aparecem erros. Explique o mecanismo do aparecimento desses erros.

2. – Sincronização entre *threads* acedendo a variáveis partilhadas, usando um *mutex* (2)

Altere o código do Problema 2 da Folha nº 6, por forma a resolver, com recurso a um *mutex* o problema de sincronização que então foi identificado.

3. – Sincronização entre *threads* acedendo a variáveis partilhadas, usando um *mutex* (3)

Modifique o programa do problema 1 desta folha, por forma a que a *thread* `verify()` execute concorrentemente com as outras *threads*. Esta *thread* só deve fazer uma verificação se o valor da sua variável local k for menor ou igual do que a variável global `pos`. Use o mesmo *mutex* para sincronizar as diferentes *threads*.

4. – Memória partilhada. Sincronização usando semáforos.

Modifique o programa do problema 1 desta folha , de modo a usar processos em vez de *threads* para preencher o *buffer* partilhado. O *buffer* deve ser criado numa região de memória partilhada e a sincronização deve ser feita com recurso a semáforos. Nota: o contador associado aos semáforos do System V tem geralmente um valor máximo muito pequeno (32767); as implementações POSIX têm geralmente valores máximos muito superiores.

5. – Problemas clássicos de sincronização: "problema dos produtores e dos consumidores" (1)

a) Escreva dois programas, executando como processos independentes, que ilustrem a solução para o "problema dos produtores e dos consumidores" estudada nas aulas teóricas. Considere que os itens produzidos são números inteiros que são colocados numa região de memória partilhada. Use semáforos do Posix System V para fazer a sincronização. Verifique que não se perdem itens produzidos nem se consomem itens não produzidos. Nota: Antes de abandonar o sistema assegure-se que as estruturas IPC criadas já não existem, usando os comandos `ipcs` e `ipcrm`.

b) Resolva o mesmo problema usando *threads* em vez de processos. Neste caso os itens produzidos devem ser guardados numa variável global.

c) Modifique o programa da alínea anterior por forma a usar *condition variables* em vez de semáforos para fazer a sincronização.

6. – Sincronização entre *threads* acedendo a variáveis partilhadas, usando uma *condition variable*

a) Resolva o "problema dos produtores e dos consumidores" usando *threads* e *condition variables* para fazer e sincronização.

b) Modifique o programa do problema 1 desta folha de modo a usar uma *condition variable* (variável de condição) para fazer a sincronização. Explique qual a vantagem do uso da *condition variable*.

7. – Problemas clássicos de sincronização: "problema dos produtores e dos consumidores" (2)

Escreva uma nova versão do programa 7 da folha 6 em que o número de *threads*, N, seja fixado à partida, sendo passado como argumento da linha de comando, transformando-o num problema do tipo produtor-consumidor. Deverá existir uma *thread* produtora que gera uma lista dos ficheiros a copiar e N *threads* consumidoras que vão copiando os ficheiros da lista. Teste o programa com diferentes números de *threads*.

8. – Filas de mensagens

Resolva o problema 8 da folha nº 5 (controlador de chegadas) usando uma fila de mensagens em vez de um *FIFO* para comunicação entre os utilizadores de tipo C e o utilizador de tipo S. NOTA: Antes de abandonar o sistema assegure-se que as estruturas *IPC* criadas já não existem, usando os comandos `ipcs` e `ipcrm`.